Università degli Studi di Roma "Tor Vergata"

Exact Algorithms for Hard Graph Problems (Algoritmi Esatti per Problemi Difficili su Grafi)

Fabrizio Grandoni

Tesi sottomessa per il conseguimento del titolo di

Dottore di Ricerca in "Informatica e Ingegneria dell'Automazione"

XVI Ciclo del Corso di Dottorato (2000-2004)

Docente Guida (Advisor): Prof. Giuseppe F. Italiano

Coordinatore (Coordinator): Prof. Daniel P. Bovet

Roma, Marzo 2004

to my little star ...

"Doh"

Homer J. Simpson

Abstract

The *vertex cover*, *independent set* and *dominating set* problems consist in determining whether an undirected graph G of n nodes admits a vertex cover, independent set and dominating set of k nodes respectively. This thesis is focused on these three fundamental NP-complete problems. For each one of them, we provide asymptotically faster (exponential-time) exact algorithms for k "sufficiently" smaller than n. We also present an improved exact algorithm for the (NP-hard) *minimum dominating set* problem, which consists in determining the minimum size of a dominating set of G.

We remark that the previously asymptotically fastest algorithm for (minimum) dominating set is the trivial enumerative algorithm.

Contents

Introduction **4**

1 Preliminaries **25**

1.1 Fast Matrix Multiplication 27

1.1.1 Fast Rectangular Matrix Multiplication 28

1.2 Vertex Cover 29

1.2.1 Kernel Reduction 31

1.2.2 Bounded Search Trees 33

1.2.3 Dynamic Programming 40

1.3 Cliques 41

1.4 Diamonds 44

1.5 Cycles 46

1.6 The Binary Constraints Satisfaction Problem 48

2 Vertex Cover **52**

2.1 A More Efficient Use of the Database 52

2.2 Branching on Connected Induced Subgraphs 54

2.3 A Further Refinement 57

3 Clique **59**

3.1 Dense Case 59

3.1.1 The (Induced) Subgraph Problem 63

3.1.2 Counting Cliques . . . 63
3.2 Sparse Case . . . 64
3.3 Diamonds . . . 66
3.3.1 Counting Diamonds . . . 68
3.4 Simple Directed 4-Cycles . . . 69
3.4.1 Remarks . . . 70

4 Decremental Clique and Applications **72**
4.1 Decremental Clique . . . 73
4.1.1 Bounds on the Complexity . . . 75
4.2 Inverse Consistency . . . 78
4.3 Max-Restricted Path Consistency . . . 81
4.3.1 Max-Restricted Path Consistency in Less Space . . . 82
4.3.2 Max-Restricted Path Consistency in Less Time . . . 87

5 Dominating Set and Set Cover **89**
5.1 Small Dominating Sets . . . 90
5.2 Minimum Dominating Set . . . 91
5.2.1 A Polynomial-Space Algorithm . . . 92
5.2.2 An Exponential-Space Algorithm . . . 94
5.2.3 Remarks . . . 96

Conclusions **97**

Acknowledgments **99**

Bibliography **101**

Introduction

A *computational problem* consists in computing, for a given input taken from a domain of *instances*, a *solution*, that is an output which satisfies a given relation with the input. For example the sorting problem consists in computing, for a given list of objects (say, integers), a permutation of the objects in non-decreasing order (according to a given order relation).

An *algorithm* is a well-defined step-by-step computational procedure which takes some inputs and provides some outputs. It can be viewed as a program which runs on a given *machine*. An algorithm solves a computational problem if, for every given instance of the problem, it computes a (correct) solution in a finite number of steps. In this thesis we will consider *deterministic* algorithms only, that is algorithms which are not able to make random choices (for a survey on *randomized algorithms*, see for example [59]).

When more than one algorithm is available to solve a given problem, we often prefer to use the most *efficient* one, that is the algorithm which uses the smallest amount of a given *resource*. The resource considered is usually *time*, measured as the number of steps required to solve the problem (what can be done in one step, depends on the machine model considered).

Since the time an algorithm takes to solve a problem may be different for each instance considered, we need a simple and easy qualitative way to summarize it. The *(worst-case) time complexity* of an algorithm is the maximum number of steps required to solve an instance of *size* n, where the size is measured as the length of a reasonably *succinct* representation of the instance. What "succinct" means, is hard to define. The following two general rules

may give an intuition:

- numbers are represented in binary (or in any other base $b > 1$);
- there is no unnecessary information added.

There are several reasons for considering the *worst-case* time complexity instead of, for example, the *average-case* one. First of all, the worst case is well defined, while defining an average case is often a difficult task (which involves the probability distribution of the instances). Secondly, the worst case analysis provides a guarantee that the algorithm will never take any longer (this information is crucial in many *critical* applications). Eventually, (nearly) worst-case instances are rather frequent in lots of practical applications (a concrete example of the well-known "Murphy's Low").

Determining the *exact* time complexity of an algorithm can be difficult, tedious and not really relevant (most of the times, all we need is its order of magnitude). For these reasons, the *asymptotic* time complexity is often preferred. An algorithm is $\mathrm{O}(f(n))$, for a given function $f(n)$ of n, if there exist positive constants C and n' such that the worst-case time complexity of the algorithm is upper bounded by $Cf(n)$ for every $n \geq n'$. An algorithm is $\Omega(f(n))$ if there exist positive constants C and n' such that the worst-case time complexity of the algorithm is lower bounded by $Cf(n)$ for every $n \geq n'$. Eventually, an algorithm is $\Theta(f(n))$ if it is $\mathrm{O}(f(n))$ and $\Omega(f(n))$.

A *polynomial-time* algorithm is an algorithm whose time complexity is $\mathrm{O}(p(n))$, for some polynomial $p(n)$ of n. All the other algorithms are usually referred as *exponential-time* algorithms. The distinction between polynomial-time and exponential-time algorithms provides a simple and neat way to separate *tractable* problems (that is, solvable in polynomial-time) from *intractable* ones. Though this distinction is not completely satisfactory (is a $\mathrm{O}(n^{1000000})$ algorithm really useful?), this kind of misbehavior almost never occurs in practice (that is, the degree of the polynomial is usually small).

Hard Problems

Now we have a notion of "hard" problems: a problem is hard if it cannot be solved in polynomial time. We already know that some important problems are not hard, since polynomial-time algorithms have been already found to solve them. Unfortunately, this is not the case for many fundamental problems. What can we say about them? Are they really hard, or do they admit polynomial-time algorithms which have not been discovered yet? Let us restrict for a moment to *decision* problems, i.e. problems whose solution is either *yes* or *no*. In particular, we will consider the class NP of the decision problems whose solution can be *certificated* in polynomial time. In other words, for each pair instance-solution, there is a certificate which allows to efficiently (polynomially) verify whether the solution provided is correct. Note that this does not imply that the solution itself can be computed in polynomial time (the problems in NP are easy "a posteriori"). The class P is the subclass of NP of the problems which are solvable in polynomial time.

An example of problem in NP is *satisfiability*: does a boolean formula admit a truth-assignment which satisfies it (that is, which makes it true). Indeed, among the NP-problems, satisfiability has a very special property. In 1971, Stephen Cook [15] showed that it is NP-*complete*. In other words, if satisfiability is solvable in polynomial time, then all the problems in NP would be polynomial-time solvable (that is $P = NP$). Later on, many other problems have been shown to be NP-complete. In fact, almost all the natural problems in NP are also NP-complete. On one hand, if one managed to solve one of these problems efficiently, he/she would automatically solve a large part of the computational problems of the world (which seems to be a good motivation). On the other hand, thousands of brilliant researchers all around the world worked on these problems for decades without finding anything better than exponential-time algorithms. For this reason, it is nowadays widely believed that NP-complete problems are not polynomial-time solvable (that is, $P \neq NP$). Determining whether $P \neq NP$ is one of the most relevant open problems of contemporary mathematics.

These negative results partially extend to problems outside NP. A problem $\mathcal{P}$ (not necessarily in NP) is *NP-hard* if any problem in NP is reducible to $\mathcal{P}$ in polynomial time (in particular, NP-complete problems are NP-hard also). Lots of natural problems outside NP are NP-hard. This is the case for example for the optimization versions of NP-complete problems. For the same reasons discussed above, NP-hard problems are not believed to be efficiently solvable.

How to Solve Hard Problems?

In previous section we observed that most of the problems of practical interest are NP-complete or NP-hard. Thus it seems very unlikely that they can be solved efficiently. Nonetheless, we still need to solve these problems somehow.

There are a few different possible approaches to the problem. A traditional way to attack hard problems is via *approximation*. Suppose that one needs to solve a NP-hard optimization problem. One may just need to find a solution which is not "too" far from the optimum (that is, an *approximated solution*). For several important problems, very efficient algorithms have been developed which provide an approximated solution very close to the (unknown) optimum. In fact, approximation algorithms is one of the most relevant and successful branch of computer science (for a survey, see for example [79]).

Unfortunately, this approach is not always satisfactory. First of all, sometimes an exact solution is needed. More interestingly, lots of negative results have been delivered concerning the *approximability* of relevant problems. For these problems, it is not possible to efficiently compute an approximated solution "very" close to the optimum (unless $P = NP$).

The problems which cannot be solved via approximation algorithms, are usually solved via *heuristic methods*. Most of the times, the instances of the hard problems which need to be solved in practice are far from the worst case instances. So it quite often happens that huge real-world instances of hard problems are quickly solved in practice. In fact, heuristic

methods are of crucial importance in the applications.

Nonetheless, like in the case of approximation algorithms, this approach is not completely satisfactory. The unique way to evaluate heuristic methods is by testing them on a selected subset of instances. First of all, there is no guarantee on their performance on the other instances (which is bad for critical applications). Moreover, the way test-instances are chosen is somehow arbitrary (or, worse, they could be selected such as to obtain good performances by a dishonest tester). Note that this is exactly the kind of drawbacks to avoid which the worst-case time complexity has been (successfully) introduced.

Exact Algorithms

There is a third approach which may help when both approximation algorithms and heuristic methods are not satisfactory: the *exact algorithms*. The aim of exact algorithms is lowering the (exponential) time complexity of hard problems.

Consider for example an algorithm $\mathcal{A}$ of time complexity 2^n which solves a given problem. Suppose that the time available to solve the problem is NT, for some (large) N, where T is the time taken by one elementary step. This seems a very reasonable assumption in most practical applications. Then $\mathcal{A}$ can solve instances of size at most $K = \log_2 n$. Suppose now that we want to solve instances of reasonably larger size, for example of size $2K$. A first possibility is running $\mathcal{A}$ on a faster machine. This does not seem a good idea since such machine should be N times faster (and we expect that N is very large in practice). A more realistic approach is trying to develop a new algorithm for the same problem of time complexity $2^{n/2}$ (computer science researchers are much cheaper than powerful computers!). Note that the existence of such (exponential-time) algorithm does not contradict the conjecture $P \neq NP$.

Enlarging the size of the worst-case instances solvable in a given amount of time is not the unique reason which makes exact algorithms interesting. In fact, while trying to lower the exponential worst-case time complexity, one is forced to exploit the combinatorial structure

of the problem in a rigorous way (since the measure of performance is rigorous). Most of the fastest polynomial-time algorithms currently used are heavily based on the theoretical results obtained in the framework of asymptotic analysis. We believe that, on long term, this will be the case for exponential-time algorithms also.

A lot of effort has been devoted in last decades to develop faster and faster exact algorithms for NP-complete and NP-hard problems, such as *maximum independent set* [5, 45, 72, 73], *vertex cover* [3, 14, 66], *(maximum) satisfiability* [4, 19, 64, 67], 3-*coloring* [6, 31] and many others. Maybe the first example is the $O(2^{n/3})$ algorithm of Tarjan and Trojanowski [77] for maximum independent set on graphs of n nodes. Note that it is significatively faster than the trivial $\Omega(2^n)$ enumerative algorithm.

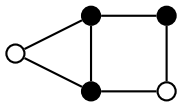
In this thesis we present improved (exponential-time) exact algorithms for three fundamental hard graph problems: *vertex cover*, *independent set* and *(minimum) dominating set* (for formal definitions, see Chapter 1). These problems have thousands of relevant applications, which range from computational biology to telecommunication networks, artificial intelligence, social sciences and so on. Indeed, they are among the most popular problems in computer science.

Vertex Covers

A *vertex cover* of an undirected graph $G = (V, E)$ of n nodes is a subset V' of nodes such that every edge in E is incident on at least one node in V'. In Figure 1 an example of vertex cover is given. The *vertex cover* problem consists in determining whether G admits a vertex cover of k nodes. The *minimum vertex cover* problem consists in determining the minimum cardinality of a vertex cover of G.

Vertex cover is one of the first problems which has been shown to be NP-complete [35]. Minimum vertex cover is NP-hard [35], approximable within a factor "slightly" smaller than 2 [51, 40, 58] and non-approximable within a factor 1.1666 [41] (unless $P = NP$). The cur-

Figure 1 The black nodes form a vertex cover, that is every edge is incident on at least one black node.

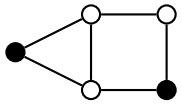


rently fastest algorithm for vertex cover for arbitrary values of k is the $O(1.1889^n)$ algorithm of Robson [73]. If k is sufficiently smaller than n, much faster algorithms exist. *Parameterized complexity* [26] is a discipline which tries to understand from which aspects of the input (the *parameter*) the hardness of problems originates. An instance of a *parameterized decision problem* is a pair (I, P), where I is the input (in the classical sense) and P, the *parameter*, is a distinguished part of the input. A problem is *fixed-parameter tractable* if its time complexity is polynomial in the size $i = |I|$ of the input, once that the size $p = |P|$ of the parameter is fixed, and if the asymptotic running time in this case is independent from the parameter. In other words, if the time complexity can be bounded by a function of the kind $f(p)i^\alpha$, where $f(\cdot)$ is an arbitrary function and α is a constant (independent of p). In 1988, Fellows [33] provided a $O(2^k n)$ algorithm for vertex cover, thus showing that vertex cover is fixed-parameter tractable (for parameter k). His algorithm is based on the *bounded search tree* technique (the basic idea is already present in a Mehlhorn's text book of 1984 [54]). A lot of effort was recently devoted to develop faster and faster algorithms for vertex cover for small values of k. Buss and Goldsmith [12] proposed a $O(2^k k^{2k+2} + kn)$ algorithm, in which they introduced the *kernel reduction* technique. Combining the bounded search tree and kernel reduction techniques, Downey and Fellows [25] derived a $O(2^k k^2 + kn)$ algorithm for vertex cover. The complexity was later reduced to $O(1.3248^k k^2 + kn)$ by Balasubramanian, Fellows and Raman [3], to $O(1.2918^k k^2 + kn)$ by Niedermeier and Rossmanith [62], to $O(1.2906^k k + kn)$ by Downey, Fellows and Stege [27], and to $O(1.2852^k k + kn)$ by Chen,

Kanj and Jia [14]. Observe that all the algorithms cited above outperform the $O(1.1889^n)$ algorithm of Robson for values of k small enough. Thanks to the *interleaving technique* of Niedermeier and Rossmanith [63], it is possible to get rid of the polynomial factor in the exponential term of the complexity. For example, the complexity of the algorithm of Chen et al. can be reduced to $O(1.2852^k + kn)$. It is worth to notice that, though such polynomial factor is not relevant from the asymptotic point of view (since the base of the exponential factor is already overestimated), it is usually indicated. The reason is that, for values of k which are not "too" big, the polynomial factors are not negligible.

Memorisation is a dynamic programming technique developed by Robson [72, 73] in the context of maximum independent set, which allows to reduce the time complexity of many exponential-time recursive algorithms at the cost of an exponential space complexity. The key-idea behind memorisation is that, if the same subproblem appears many times, it may be convenient to store its solution instead of recomputing such solution from scratch. With this technique Robson managed to derive a $O(1.1889^n)$ exponential-space algorithm for maximum independent set [73] from his own $O(1.2025^n)$ polynomial-space algorithm for the same problem [72]. Memorisation cannot be applied to the algorithm of Chen et al. This happens because their algorithm branches on subproblems involving graphs which are not induced subgraphs of the original graph. Niedermeier and Rossmanith [65, 66] applied memorisation to their $O(1.2918^k k^2 + kn)$ polynomial-space algorithm for vertex cover, thus obtaining a $O(1.2832^k k^{1.5} + kn)$ exponential-space algorithm for the same problem. This is also the currently fastest algorithm for vertex cover for small values of k. It is worth to notice that memorisation is not compatible with the interleaving technique of Niedermeier and Rossmanith. In fact, the interleaving technique is based on the idea that most of the subproblems concern graphs with few nodes. This is not the case when memorisation is applied.

Figure 2 The black nodes form an independent set, that is the black nodes are not adjacent.



Our Results

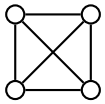
The kind of memorisation which is currently applied to vertex cover is in some sense a weaker version of the technique originally proposed by Robson for maximum independent set. This is mainly due to the structural differences between the two problems. In Chapter 2 we present a simple technique which allows to get rid of these structural differences, thus allowing to apply memorisation to vertex cover in its full strength. By applying this refined technique to the $O(1.2918^k k^2 + kn)$ algorithm of Niedermeier and Rossmanith, we obtain a $O(1.2759^k k^{1.5} + kn)$ exponential-space algorithm for vertex cover. With a further refined technique, we reduce the complexity to $O(1.2745^k k^4 + kn)$.

Independent Sets and Cliques

An *independent set* of an undirected graph $G = (V, E)$ of order n is a subset V' of pairwise non-adjacent nodes. In Figure 2 an independent set of a graph is depicted. The *independent set* problem consists in determining whether G admits an independent set of k nodes. The *maximum independent set* problem consists in determining the maximum cardinality of an independent set of G. The (maximum) independent set and (minimum) vertex cover problems are strictly related. In particular, $V' \subseteq V$ is a vertex cover of G if and only if $V \backslash V'$ is an independent set of G.

Independent set problem is NP-complete [35]. Maximum independent set problem is NP-hard [35] and hard to approximate [42]. There is a trivial $O(2^n n^2)$ algorithm for

Figure 3 A clique of four nodes.



maximum independent set. This algorithm simply enumerates and checks all the subsets of nodes. Many faster (exponential-time) algorithms for this problem have been developed [5, 45, 72, 77]. The currently fastest algorithm for maximum independent set is the $O(1.1889^n)$ algorithm of Robson [73].

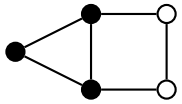
It is not known whether independent set with parameter k is fixed-parameter tractable. Indeed, this is not believed to be true. Downey and Fellows [25, 26] delivered completeness results for independent set. If one could show that independent set is fixed-parameter tractable for parameter k, then, by reduction, this would be the case for many other parameterized problems. More precisely, let FPT be the family of the fixed-parameter tractable problems. Downey and Fellows showed that independent set with parameter k is complete for the complexity class $W[1]$, where:

$$FPT \subseteq W[1].$$

It is conjectured that $FPT \neq W[1]$, which would imply that independent set with parameter k is not fixed-parameter tractable. Unfortunately, this conjecture is far from being proved (it would imply $P \neq NP$).

A *clique* is an undirected graph such that its nodes are pairwise adjacent. A k-clique is a clique of k nodes. In Figure 3 a 4-clique is depicted. Graph G *contains* a k-clique if there is a subset V' of k nodes such that the graph $G[V']$ induced on G by V' is a clique. A triangle (3-clique) contained in a graph is depicted in Figure 4. The *clique problem* consists in determining whether G contains a k-clique. The *maximum clique problem* consists in

Figure 4 The black nodes induce a triangle (3-clique).



determining the maximum order of a clique contained in G.

Clique and independent set problems are strictly related. In particular, G contains a k-clique if and only if the *complement* $\overline{G}$ of G has an independent set of size k. It follows that clique problem is NP-complete and $W[1]$-complete for parameter k. Moreover, maximum clique problem is NP-hard and hard to approximate.

The currently fastest algorithm for maximum clique problem is the $O(1.1889^n)$ algorithm of Robson [73]. Itai and Rodeh [44], showed how to *detect* a triangle (3-clique) in $O(n^\omega)$ time, where $\omega < 2.376$ [17] is the exponent of fast square matrix multiplication. Nešetřil and Poljak [61] generalized the algorithm of Itai and Rodeh to the detection of cliques of arbitrary (fixed) order k. Their algorithm has time complexity $O(n^{\alpha(k)})$, where $\alpha(k) = \lfloor k/3 \rfloor \omega + k \pmod 3$. This algorithm outperforms the $O(1.1889^n)$ algorithm of Robson [73] for small values of k. Alon, Yuster and Zwick [2] showed how to detect a triangle in $O(e^{\frac{2\omega}{\omega+1}})$ time, where e denotes the number of edges of G. Kloks, Kratsch and Müller [49] generalized the result of Alon et al. to cliques of arbitrary order. The running time of their algorithm is $O(e^{\frac{\alpha(k)\alpha(k-1)}{\alpha(k)+\alpha(k-1)-1}})$. If $k \pmod 3 \neq 0$, their running time is $O(e^{\alpha(k)/2})$, which is never inferior to the running time obtained by Nešetřil and Poljak for *dense* graphs. This does not hold when $k \pmod 3 = 0$. In that case the $O(n^{\alpha(k)})$ algorithm is faster if G is dense enough. Kloks et al. posed the question [49] whether there exists a $O(e^{\alpha(k)/2})$ algorithm for the detection of cliques of arbitrary order $k \geq 3$.

Figure 5 The graph on the right contains the graph on the left as an induced subgraph (and thus as a subgraph).

Figure 6 The graph on the right contains the graph on the left as a subgraph but not as an induced subgraph.

Other (Induced) Subgraphs

Given two undirected graphs F and G, G *contains* the (induced) subgraph F if F is isomorphic to an (induced) subgraph of G. Examples of subgraphs and induced subgraphs are depicted in Figure 5 and 6. The *(induced) subgraph problem* consists in determining whether an undirected graph G of n nodes contains an (induced) subgraph F of order k. Nešetřil and Poljak [61] provided a reduction from the (induced) subgraph problem to the clique problem. This reduction allows to detect an (induced) subgraph of k nodes, for every fixed k, in the same time required to detect a clique of the same order. In particular, this can be done in $\mathrm{O}(n^{\alpha(k)})$ time.

Faster algorithms exist for some classes of (induced) graphs. Kloks et al. [49] investigated the detection of induced graphs of order four. In particular, they considered the detection of induced *diamonds*. A *diamond*, which is depicted in Figure 7, is obtained from a 4-clique by removing one edge. Detecting diamonds is important since maximum clique is solvable in polynomial time in diamond-free graphs [78]. Kloks et al. showed that a diamond can be

Figure 7 A diamond.

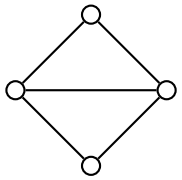
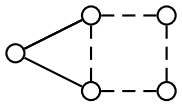


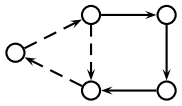
Figure 8 A simple undirected cycle of length four is pointed out via dashed lines.



detected more efficiently than a clique of the same order, namely in $O(n^\omega + e^{3/2})$ steps.

An *(undirected) cycle* of G is a sequence $v_0, v_1 \dots v_{k-1}$ of nodes such that, for every $i \in \{0, 1 \dots k-1\}$, $\{v_i, v_{i+1 \pmod k}\} \in E$. A *(directed) cycle* of a directed graph $G = (N, A)$ is a sequence $v_0, v_1 \dots v_{k-1}$ of nodes such that, for every $i \in \{0, 1 \dots k-1\}$, $(v_i, v_{i+1 \pmod k}) \in E$. Both in the directed and in the undirected case, the *length* of the cycle is the number of its nodes, and a cycle is *simple* if no node is repeated in the sequence. A k-cycles is a cycle of length k. A C_k is a simple k-cycle. Examples of undirected and directed simple cycles are depicted in Figures 8 and 9 respectively. The detection of a C_k, both in the directed and in the undirected case, is one of the most natural problems in algorithmic graph theory. Though the problem is easily stated, fast algorithms to solve it are far from being obvious and progress in terms of faster algorithms has been constantly reported in the last decades. Itai and Rodeh [44] developed two algorithms to detect a C_3 in $O(n^\omega)$ and $O(e^{1.5})$ time respectively. Alon, Yuster and Zwick [1] described an algorithm which detects a C_k, for every fixed k, in $O(n^\omega)$ expected time and $O(n^\omega \log n)$ worst case time. They moreover developed algorithms [2] which detect a C_k in $O(e^{2-1/\lceil k/2 \rceil})$ time and a C_3 in $O(e^{2\omega/(\omega+1)})$

Figure 9 A simple directed cycle of length three is pointed out via dashed lines. Note that the graph does not contain simple directed cycles of length four.



time. The authors pose the question, whether fast matrix multiplication can also be used to speed up their techniques to detect a C_k for $k \geq 4$. All of the above methods work in the directed as well as in the undirected case. Even cycles in undirected graphs can be found even faster. Yuster and Zwick [80] showed that an even undirected cycle can be found in $O(n^2)$ steps. Alon, Yuster and Zwick [2] showed that a directed C_{4k-2} and a directed C_{4k} can be detected in $O(e^{2-\frac{k+1}{2k^2}})$ and $O(e^{2-\frac{1}{k}+\frac{1}{2k+1}})$ time respectively.

Our Results

In Chapter 3 we present our results concerning the detection of cliques and of other (induced) subgraphs. In Sections 3.1 and 3.2 we provide faster algorithms for the detection of cliques of fixed order k in sparse and dense graphs respectively. Our algorithm for dense graphs runs in time $O(n^{\beta(k)}) = O(n^{\omega(\lfloor k/3 \rfloor, \lceil (k-1)/3 \rceil, \lceil k/3 \rceil)})$, where $O(n^{\omega(r,s,t)})$ is the running time of the multiplication of a $n^r \times n^s$ matrix by a $n^s \times n^t$ matrix. In the sparse case, our algorithm runs in $O(e^{\beta(k)/2})$ steps, for every fixed $k \geq 6$. This means an improvement over the fastest methods for dense and sparse graphs in the case that $k \equiv 1 \pmod 3$ and $4 \leq k \leq 16$ as well as that $k \equiv 2 \pmod 3$ and $k \geq 5$. In addition, for sparse graphs we obtain faster running times for $k \equiv 0 \pmod 3$ when $k \geq 6$. This also gives a partially positive answer to the problem posed by Kloks, Kratsch and Müller in [49]. A comparison of the running times of the previous best algorithms and our results for $4 \leq k \leq 7$ is depicted in Table 1.

In Section 3.3 we present an improved algorithm for diamonds detection. Our algorithm

k	Previous best [49, 61]	This thesis
4	$O(n^{3.376})$, $O(e^{1.688})$	$O(n^{3.334})$, $O(e^{1.682})$
5	$O(n^{4.376})$, $O(e^{2.188})$	$O(n^{4.220})$, $O(e^{2.147})$
6	$O(n^{4.751})$, $O(e^{2.559})$	$O(e^{2.376})$
7	$O(n^{5.751})$, $O(e^{2.876})$	$O(n^{5.714})$, $O(e^{2.857})$

Table 1: Running time comparison for clique problem.

has a $O(e^{3/2})$ time complexity and does not require fast matrix multiplication subroutines (thus being faster and easier to implement than the algorithm of Kloks et al. [49]).

In Section 3.4 we eventually give a partially positive answer to the question posed by Alon, Yuster and Zwick in [2], by providing a $O(n^{1/\omega}e^{2-2/\omega})$ algorithm for the detection of directed C_4. This bound is asymptotically smaller than the previous best bounds $O(n^{\omega})$ and $O(e^{1.5})$ for $\alpha \in \left(\frac{2}{4-\omega}, \frac{\omega+1}{2}\right)$, where $e = n^{\alpha}$. Part of the ideas at the base of our algorithm have been recently exploited by Yuster and Zwick [81] to develop faster algorithms for cycle detection in sparse graphs.

Decremental Clique and Constraint Programming

There is a wealth of research on *dynamic graph problems*, which consist in checking a given property on graphs subject to dynamic changes, such as deletions or insertions of nodes or edges [23, 24, 34, 47, 48, 75, 82]. A *dynamic graph algorithm* is an algorithm which answers queries about a given property of the graph, during such kind of dynamic changes. If only deletions or insertions are allowed, the dynamic problem is also called *decremental* or *incremental* respectively.

The *decremental clique problem* consists in dynamically determining whether an undirected graph G of order n contains a clique of order k, during deletions of nodes. This problem naturally arises from filtering techniques for the *binary constraints satisfaction problem*. To the best of our knowledge, no non-trivial algorithm is known for the *decremental clique problem*, while several non-trivial results are available for its static version (as previously

discussed).

The Binary Constraints Satisfaction Problem

Constraint programming [53] is a declarative programming paradigm which allows one to naturally formulate computational problems. A computational problem is formulated as a *constraint satisfaction problem*, which consists in determining whether there is an instantiation of a set of variables, defined on finite domains, which satisfies a set of constraints. Any such instantiation is a *solution* for the *constraints network*. The task of the constraint programming system is to find a solution (or alternatively all the solutions, or the "best" one).

The development of efficient constraint programming systems leads to interesting algorithmic questions as stressed in [55]. A reason is that many related problems can be mapped into equivalent problems on graphs, for which powerful algorithmic techniques are available. Consider for example the well studied *alldifferent* constraint, which requires that pairwise distinct values are assigned to a set X of variables. Regin [71] observed that the satisfiability of this constraint is equivalent to a matching problem in the following bipartite graph. On the left side there is a node i for each variable $i \in X$, and on the right side there is a node a for each value which the variables in X may take (without repetitions). A node i on the left side is adjacent to a node a on the right side if and only if a belongs to the domain of i. The constraint is satisfiable if and only if there is a matching in which all the nodes on the left side are matched. Several non-trivial filtering algorithms have been developed to deal with the *alldifferent constraint* [36, 56, 68, 71], as well as for other constraints of great practical and theoretical interest such as the *dominance constraint* [50] and the *sortedness constraint* [39, 56]. For a comprehensive classification of constraints using graph terminology, see [7].

One of the basic problems in constraint programming is the *binary constraint satisfac-*

tion problem, which consists in determining whether there is an instantiation of a set of n variables, defined on domains of cardinality at most d, which satisfies a set of e binary constraints. Any such instantiation is a *solution* for the *binary constraints network* considered. The assignment (i, a) of value a to variable i is *consistent* if there is a solution which assigns a to i, and *inconsistent* otherwise. Inconsistent *value assignments* can be removed from the network without loosing any solution (by *removing* a value assignment (i, a), we mean removing the value a from the domain of variable i). It is not known whether a polynomial-time algorithm exists to detect all the inconsistent value assignments. Indeed, this is not believed to be true since the binary constraints satisfaction problem is NP-complete. For this reason, *filtering algorithms* have been developed, which allow to "quickly" remove a subset of the inconsistent value assignments. Most of them are based on some kind of *local consistency* property. Consider a property $\mathcal{P}$ ("easy" to check) which all the consistent value assignments need to satisfy (*local consistency property*). Notice that the condition is only necessary: a value assignment may satisfy $\mathcal{P}$ while being inconsistent. If a value assignment does not satisfy $\mathcal{P}$, one can remove it from the network without removing any solution. This process can be clearly iterated. If, during the process, one of the domains becomes empty, the binary constraints network admits no solution. In fact, every solution needs to assign exactly one value to each variable.

One of the most important filtering properties [30] is *ℓ-inverse consistency*. This property is also called *arc consistency* if $\ell = 2$ [52], and *path inverse consistency* if $\ell = 3$ [30]. The fastest ℓ-inverse consistency based filtering algorithm has a $O(n^\ell d^\ell)$ time complexity [20].

In [21], a new and promising local consistency property has been proposed: the *max-restricted path consistency*. Computational experiments give evidence that max-restricted path consistency offers a particularly good compromise between computational cost and pruning efficiency [22]. Debruyne and Bessiere [21] developed the fastest known filtering algorithm based on max-restricted path consistency, denoted by `max-RPC-1`, which has a

$O(end^3)$ time complexity and a $O(end)$ space complexity.

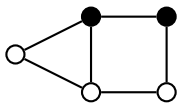
Our Results

In Chapter 4 we present our results concerning the decremental clique problem and its application to filtering for constraint programming.

In Section 4.1 we provide an algorithm to dynamically maintain the number $K_k(v)$ of k-cliques in which each node v of an undirected graph G is contained, during deletions of nodes. The *preprocessing time* of this algorithm is $O(n^{\beta(k)})$, where $\beta(k) = \omega(\lfloor k/3 \rfloor, \lceil (k-1)/3 \rceil, \lceil k/3 \rceil)$. The *query time* is $O(1)$ and the amortized *update time* per deletion is $O(n^{\widetilde{\beta}(k)}) = O(n^{\beta(k)-0.8})$. This algorithm can be easily adapted such as to solve the decremental clique problem. In fact, the answer to the decremental clique problem is *yes* if and only if at least one of the (updated) values $K_k(v)$ is greater than zero.

This decremental algorithm naturally applies to *filtering* for the binary constraints satisfaction problem. In particular, we show how to speed up the filtering based on inverse consistency and max-restricted path consistency. In Section 4.2, we present a $O(n^\ell d^{\widetilde{\beta}(\ell)+1})$ filtering algorithm based on ℓ-inverse consistency. This improves on the $O(n^\ell d^\ell)$ algorithm of Debruyne [20] for every $\ell \geq 3$. In Section 4.3, we present a filtering algorithm based on max-restricted path consistency, with a $O(end^{2.575})$ time complexity and a $O(end^2)$ space complexity. In the same section we also show another max-restricted path consistency based filtering algorithm which does not make use of the decremental algorithm of Section 4.1. It has the same time complexity as the algorithm of Debruyne and Bessiere [21], which is $O(end^3)$, but it has a smaller space complexity, that is $O(ed)$. The experiments performed on *random consistency graphs* suggest that this second algorithm is faster than the algorithm of Debruyne and Bessiere for problems of low *density* and high or low *tightness*.

Figure 10 The black nodes form a dominating set, that is the nodes of the graph are either adjacent to the black nodes or belong to the black nodes.



Dominating Sets and Set Covers

A *dominating set* of an undirected graph $G = (V, E)$ is a subset V' of nodes such that any node which is not contained in V' is adjacent to at least one node in V'. In Figure 10 an example of dominating set is depicted. The *dominating set* problem consists in determining whether G admits a vertex cover of k nodes. The *minimum dominating set* problem consists in determining the minimum cardinality of a dominating set of G.

Dominating set problem is NP-complete [35]. Minimum dominating set problem is NP-hard [35] and hard to approximate [69]. The fastest algorithm known to solve minimum dominating set is the $O(2^n n^2)$ trivial algorithm which enumerates and checks all the subsets of nodes. It is not known whether dominating set is fixed-parameter tractable for parameter k. Moreover, as in the case of independent set, this is not believed to be true. Downey and Fellows [25, 26] showed that dominating set with parameter k is complete for the complexity class $W[2]$, where:

$$W[1] \subseteq W[2].$$

It is conjectured that $FPT \neq W[1]$. This would imply that dominating set is not fixed-parameter tractable for parameter k. The fastest known algorithm to solve dominating set for small values of k is the $O(n^{k+1})$ trivial algorithm which enumerates all the subsets of k nodes and tests whether one of these subsets is a dominating set. Regan [70] posed the question, whether there exists an algorithm which is faster than the trivial one.

Let $\mathcal{S}$ be a collection of subsets of a given *universe* $\mathcal{U}$. A *set cover* of $\mathcal{U}$ is a subset $\mathcal{S}'$ of $\mathcal{S}$ such that every element s of $\mathcal{U}$ is contained in at least one set S' of $\mathcal{S}'$. Without loss of generality, we assume that $\mathcal{S}$ *covers* $\mathcal{U}$:

$$\bigcup_{S \in \mathcal{S}} S = \mathcal{U}.$$

The *minimum set cover* problem consists in determining the minimum cardinality of a set cover of $\mathcal{S}$. Minimum set cover is NP-hard [35], approximable within $(1 + \ln |\mathcal{U}|)$ [46] and not approximable within $c \log |\mathcal{S}|$ for some $c > 0$ [69]. At the best of our knowledge, the fastest algorithm for minimum set cover is the trivial $\Omega(2^{|\mathcal{S}|})$ algorithm which enumerates and checks all the subsets of $\mathcal{S}$.

Note that minimum dominating set can be naturally formulated as a minimum set cover problem, where there is a set $N[v] = N(v) \cup \{v\}$ for each node $v \in V$.

Our Results

Our results concerning dominating set are described in Chapter 5. In Section 5.1 we provide an improved algorithm for dominating set for small values of k. The running time of our algorithm is $\mathrm{O}(n^{\omega(\lfloor k/2 \rfloor, 1, \lceil k/2 \rceil)}) = \mathrm{O}(n^{k+\omega-2})$. This improves on the trivial algorithm for any fixed $k \geq 2$, thus answering to the question posed by Regan in [70]. A comparison of our algorithm and the trivial one is shown in Table 2 for $2 \leq k \leq 7$. For $k \geq 8$, the complexity

k	Previous best	This thesis
2	$\mathrm{O}(n^3)$	$\mathrm{O}(n^{2.376})$
3	$\mathrm{O}(n^4)$	$\mathrm{O}(n^{3.334})$
4	$\mathrm{O}(n^5)$	$\mathrm{O}(n^{4.220})$
5	$\mathrm{O}(n^6)$	$\mathrm{O}(n^{5.220})$
6	$\mathrm{O}(n^7)$	$\mathrm{O}(n^{6.063})$
7	$\mathrm{O}(n^8)$	$\mathrm{O}(n^{7.063})$

Table 2: Running time comparison for dominating set.

of our algorithm is $\mathrm{O}(n^{k+o(1)})$.

In Section 5.2 we present a $O(1.3424^d)$ algorithm for minimum set cover, where d is the *dimension* of the problem, that is the sum of the number $|\mathcal{S}|$ of sets available and of the number $|\mathcal{U}|$ of elements which need to be covered. The dimension of the minimum set cover formulation of minimum dominating set is $d = 2n$ (there is one set for each node and the set of elements which need to be covered is the set of nodes). It follows that minimum dominating set can be solved in $O(1.3424^{2n}) = O(1.8021^n)$ time.

Outline of the Thesis

The rest of this thesis is organized as follows. In Chapter 1 we introduce some notation and preliminary results. In Chapter 2, 3, 4 and 5 we present our improved algorithms for vertex cover, clique, decremental clique and dominating set respectively. Chapter 3 also contains our results concerning cycles and diamonds detection. Our results concerning filtering for constraint programming are described in Chapter 4.

Part of the results of this thesis already appear in the literature. The algorithms for the detection of small dominating sets and cliques, 4-cycles and diamonds, a joint work with Friedrich Eisenbrand, appear in [28] and [29]. The algorithms for max-restricted path consistency based filtering, a joint work with Giuseppe F. Italiano, appear in [38]. Other results are currently under consideration for possible publication [13, 37].

Chapter 1

Preliminaries

In this chapter we provide some preliminary notions which will be used in the following chapters.

We assume that the reader is familiar with elementary notions concerning algorithms and complexity. We use standard sets and graphs notation as, for example, in [18]. An *(undirected) graph* G is a pair (V, E) where V is a set and E is a set of subsets of V of cardinality two. The elements of V and E are called *nodes* (or *vertices*) and *edges* respectively. Let $n = |V|$ and $e = |E|$. The *order* of G is n and its *size* is $n + e$. We often implicitly assume that $n = \mathrm{O}(e)$. Moreover, without loss of generality, we sometimes assume $V = \{1, 2 \dots n\}$. Two nodes u and v are *adjacent* if there is an edge $\{u, v\} \in E$. In that case, we say that $\{u, v\}$ is *incident* on u and v. The *neighborhood* $N(v)$ of node v is the set of nodes adjacent to v. We sometimes denote by $N[v]$ the set $N(v) \cup \{v\}$. The *degree* $deg(v)$ of v is the cardinality of $N(v)$. A graph is *complete* if each pair of distinct nodes is adjacent. A graph is *dense* if e is "near" to the maximum possible (which is $n(n-1)/2$), and *sparse* otherwise. The meaning of "near" depends on the applications. The *adjacency matrix* A of G is a 0-1 matrix such that, for each pair of nodes v and w, $A[v, w] = 1$ if and only if v and w are adjacent (in particular A is symmetric and the main diagonal is set to zero). A *walk* of *length* k is a sequence $v_1, v_2 \dots v_k$ of k nodes such that $\{v_i, v_{i+1}\} \in E$, for every $i \in \{1, 2 \dots k-1\}$. A *path* is a walk where no node is repeated. A *cycle* is a walk $v_1, v_2 \dots v_k$ with the extra

condition that $\{v_k, v_1\} \in E$. A cycle is *simple* if no node is repeated. A simple cycle of length k is denoted by C_k.

Let $F = (V', E')$ and $G = (V, E)$ be two undirected graphs. The *complement* $\overline{G}$ of G is a graph $(\overline{V}, \overline{E})$ such that $\overline{V} = V$ and, for every pair of distinct nodes $\{u, v\}$, $\{u, v\} \in E$ if and only if $\{u, v\} \notin \overline{E}$. The graph F is a *subgraph* of graph G if $V' \subseteq V$ and $E' \subseteq E$. The graph F is an *induced subgraph* of graph G is F is a subgraph of G and two nodes of F are adjacent if and only if they are adjacent in G. Note that the induced subgraphs of G are univocally determined by giving a subset V' of V. For a given $V' \subset V$, we denote the corresponding induced subgraph of G by $G[V']$, and we say that V' *induces* $G[V']$ on G. An *isomorphism* between F and G is a bijection between their vertex sets which preserves adjacency (that is, two nodes of F are adjacent if and only if the corresponding nodes of G are adjacent). If such isomorphism exists, the graphs F and G are *isomorphic*. Graph G *contains* the (induced) subgraph F if there is an (induced) subgraph F' of G isomorphic to F.

A p-partite graph $G = (\{V_1, V_2 \dots V_p\}, E)$ is a graph where the set of nodes is $V = \bigcup_i V_i$, the set of edges is E, the subsets V_i (*partitions*) are disjoint, and the nodes in the same partition are not adjacent. If $p = 2$, the graph is *bipartite*. A p-partite graph is *complete* if each pair of nodes taken from distinct partitions are adjacent.

A *directed graph* G is a pair (N, A) where N is a set of *nodes* and $A \subseteq N \times N$ is a set of *arcs* (or *directed edges*). Most of the definitions concerning undirected graphs naturally extend to directed graphs. Where not otherwise specified, by "graph" we mean "undirected graph". In particular, directed graphs will be considered only in the sections concerning cycles detection (Sections 1.5 and 3.4).

1.1 Fast Matrix Multiplication

We assume that the reader is familiar with matrices notation. Given a matrix M, by $M[i, j]$ we denote the element of M in the i-th row and j-th column. Given a $r \times p$ matrix M and a $p \times c$ matrix N, the *product* of M and N is a $r \times c$ matrix $MN = M \cdot N$ such that, for every $i \in \{1, 2 \dots r\}$ and for every $j \in \{1, 2 \dots c\}$:

$$MN[i, j] = \sum_{k=1}^{p} M[i, k]N[k, j]. \tag{1.1}$$

Consider the cost of *multiplying* two $n \times n$ matrices. By simply applying equation (1.1), one obtains a $O(n^3)$ algorithm. Surprisingly, asymptotically faster square matrix multiplication algorithms exist. In 1969, Strassen [76] presented the following $O(n^{2.81})$ algorithm for square matrix multiplication. For simplicity, let us assume that $n = 2^k$ for some positive integer k. The generalization to arbitrary values of n is straightforward (for example, one can add rows and columns made of zeros until n becomes a power of two; this way the dimension is at most doubled). If $n = 1$, the algorithm returns the (scalar) product of A and B. Otherwise, the algorithm decomposes matrices A and B in four $\frac{n}{2} \times \frac{n}{2}$ blocks each as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Then it computes the following matrices:

$$D_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$D_2 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$D_3 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$$

$$D_4 = (A_{11} + A_{12}) \cdot B_{22}$$

$$D_5 = (A_{21} + A_{22}) \cdot B_{11}$$

$$D_6 = A_{11} \cdot (B_{12} - B_{22})$$

$$D_7 = A_{22} \cdot (B_{21} - B_{11})$$

The matrix multiplications required to compute the D_i are performed recursively. The algorithm eventually returns:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} D_1 + D_2 - D_4 + D_7 & D_4 + D_6 \\ D_5 + D_7 & D_1 - D_3 - D_5 + D_6 \end{bmatrix}.$$

Let $T(n)$ be the number of (scalar) sums and multiplications performed by this algorithm. The following relation holds:

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1; \\ 18n^2 + 7T(n/2) & \text{otherwise.} \end{cases}$$

This implies that $T(n)$ is $\mathrm{O}(n^{\log_2 7}) = \mathrm{O}(n^{2.81})$.

After the seminal work of Strassen, several faster and faster algorithms have been developed. For a comprehensive treatment, see for example [11]. Some of the most significative improvements achieved are shown in Table 3. The currently fastest algorithm for square matrix multiplication is the $\mathrm{O}(n^{2.376})$ algorithm of Coppersmith and Winograd [17].

Authors	Year of Discovery	Complexity
Strassen	1968	$\mathrm{O}(n^{2.808})$
Pan	1978	$\mathrm{O}(n^{2.781})$
Pan	1979	$\mathrm{O}(n^{2.522})$
Coppersmith, Winograd	1980	$\mathrm{O}(n^{2.498})$
Strassen	1986	$\mathrm{O}(n^{2.479})$
Coppersmith, Winograd	1986	$\mathrm{O}(n^{2.376})$

Table 3: Time complexity of some square matrix multiplication algorithms.

1.1.1 Fast Rectangular Matrix Multiplication

By $\mathrm{O}(n^{\omega(r,s,t)})$ we denote the time required to multiply a $n^r \times n^s$ matrix by a $n^s \times n^t$ matrix. It turns out that, for all the fastest rectangular matrix multiplication algorithms known,

$$\omega(r, s, t) = \omega(r', s', t'),$$

for any permutation (r', s', t') of the triple (r, s, t). Moreover,

$$\omega(r, s, t) = z\omega(r/z, s/z, t/z),$$

for any fixed $z > 0$.

A rectangular matrix multiplication can be executed through a straightforward decomposition into square blocks and fast square matrix multiplication. In other words:

$$\omega(r, s, t) = r + s + t + (\omega - 3)\min\{r, s, t\}.$$

Faster algorithms are available. The currently fastest rectangular matrix multiplication algorithms available are due to Coppersmith, Huang and Pan [16, 43]. The following bounds hold. For every $r \leq 1$:

$$\omega(1, 1, r) \leq \begin{cases} 2 + o(1) & \text{if } 0 \leq r \leq \alpha = 0.294; \\ \omega - (1 - r)\frac{\omega - 2}{1 - \alpha} & \text{if } \alpha < r \leq 1. \end{cases} \tag{1.2}$$

For every $0 \leq t \leq 1 \leq r$:

$$\omega(t, 1, r) \leq \begin{cases} r + 1 + o(1) & 0 \leq t \leq \alpha; \\ r + 1 + (t - \alpha)\frac{\omega - 2}{1 - \alpha} + o(1) & \alpha < t \leq 1. \end{cases} \tag{1.3}$$

1.2 Vertex Cover

A *vertex cover* of an undirected graph $G = (V, E)$ of n nodes is a subset V' of nodes such that every edge in E is incident on at least one node in V'. The *vertex cover* problem consists in determining whether G admits a vertex cover of k nodes. The *minimum vertex cover* problem consists in determining the minimum cardinality $mvc(G)$ of a vertex cover of G.

Lemma 1.2.1 *Let $G = (V, E)$ be an undirected graph. The following properties hold:*

1. *No minimum vertex cover contains a node of degree zero.*
2. *For any given node v, every vertex cover contains v or $N(v)$.*
3. *If there is a node v of degree one, $N(v) = \{w\}$, there is a minimum vertex cover which contains w.*
4. *If there is a node v of degree two, $N(v) = \{u, w\}$, there is a minimum vertex cover which contains both u and w or neither of them.*

Proof.

(1) Suppose that V' is a minimum vertex cover of G which contains a node v of degree zero. Then $V' \backslash \{v\}$ is a vertex cover of cardinality strictly smaller than V', which is a contradiction.

(2) Suppose that V' contains neither v nor a node $w \in N(v)$. Thus the edge $\{v, w\}$ is not incident on any node of V', which is a contradiction.

(3) Let V' be a minimum vertex cover of G which contains a node v of degree one, $N(v) = \{w\}$. Note that V' cannot contain w since it is a minimum vertex cover. By replacing v with w in V', one obtains a vertex cover of the same cardinality (and thus minimum) which contains w.

(4) Let V' be a minimum vertex cover which contains node u and does not contain node w, where $N(v) = \{u, w\}$ for some other node v. Thus V' needs to contain v. By replacing v with w in V', we obtain a vertex cover of the same cardinality (and thus minimum) which contains both u and w. □

An *independent set* of an undirected graph $G = (V, E)$ of n nodes is a subset V' of nodes which are pairwise not adjacent. The *independent set* problem consists in determining whether G admits an independent set of k nodes. The *maximum independent set* problem consists in determining the maximum cardinality $mis(G)$ of an independent set of G.

The vertex cover and independent set problems are strictly related. In particular, the following relation holds.

Proposition 1.2.1 *A set $V' \subseteq V$ is a vertex cover of an undirected graph $G = (V, E)$ if and only if the set $V'' = V \backslash V'$ is an independent set of G.*

Proof. Let V' be a vertex cover of G. Suppose by contradiction that $V'' = V \backslash V'$ is not an independent set of G. Then there is an edge $\{u, v\} \in E$ such that both u and v are in V''. Thus neither u nor v is in V', which contradicts the fact that V' is a vertex cover.

Let V'' be an independent set of G. Suppose by contradiction that $V' = V \backslash V''$ is not a vertex cover of G. Then there is an edge $\{u, v\} \in E$ such that neither u nor v is in V'. Thus both u and v are in V'', which contradicts the fact that V'' is an independent set. □

This simple result has a straightforward corollary.

Corollary 1.2.1 *Let G be an undirected graph of n nodes. The following relation holds:*

$$mvc(G) = n - mis(G).$$

This implies that every algorithm to solve maximum independent set can be easily adapted to solve minimum vertex cover and viceversa. In particular, the exponential-space $O(1.1889^n)$ algorithm of Robson [73] for maximum independent set can be used for this purpose.

The fastest algorithm known for small values of k is the $O(1.2832^k k^{1.5} + kn)$ exponential-space algorithm of Niedermeier and Rossmanith [65, 66]. The currently fastest algorithms for vertex cover for small values of k are based on three basic techniques: *kernel reduction*, *bounded search trees* and *dynamic programming*.

1.2.1 Kernel Reduction

Let (G, k) be an instance of vertex cover. The idea behind *kernel reduction* is to reduce (in polynomial time) the original problem (G, k) to an equivalent problem (G', k'), where $k' \leq k$ and the size of G', the *kernel*, is a function of k only.

Buss and Goldsmith [12] showed how to obtain a kernel with $O(k^2)$ nodes and edges in $O(kn)$ time. Suppose that G contains a node v of degree greater than k. For Property (2) of Lemma 1.2.1, every vertex cover of G contains either v or $N(v)$. This implies that all the vertex covers of G of cardinality at most k must contain node v. In other words (G, k) is equivalent to $(G[V \backslash \{v\}], k - 1)$. By iterating this reasoning, one obtains (in $O(kn)$ time) a new problem (G', k') equivalent to the original one, where $k' \leq k$ and G' is an induced subgraph of G whose nodes have degree at most k'. Note that we can assume that G' does not contain nodes of degree zero, since they can be filtered out for Property (1) of Lemma 1.2.1.

If G' contains more than $k'k$ edges, it cannot have a vertex cover of cardinality at most k. Thus either the solution of (G', k') is trivial, or G' contains $O(k^2)$ edges (and thus $O(k^2)$ nodes).

Proposition 1.2.2 *Let (G, k) be an instance of vertex cover, where the order of G is n. The algorithm above, of running time $O(kn)$, solves (G, k) or reduces it to an equivalent problem (G', k'), where $k' \leq k$ and G' contains $O(k^2)$ nodes and edges.*

Chen, Kanj and Jia [14] showed how to further reduce the number of nodes in the kernel to at most $2k$, with an extra $O(k^3)$ cost. They use the following result of Nemhauser and Trotter [60].

Theorem 1.2.1 *There is an $O(\sqrt{n}e)$ time algorithm that, given an undirected graph $G = (V, E)$ with $n = |V|$ and $e = |E|$, computes two disjoint subsets $A(V)$ and $B(V)$ of V such that:*

1. *$mvc(G) = mvc(G[A(V)]) + |B(V)|$;*
2. *$mvc(G[A(V)]) \geq |A(V)|/2$.*

The kernel reduction algorithm of Chen et al. works as follows. They first apply the $O(kn)$ algorithm of Buss and Goldsmith to the original problem (G, k), thus obtaining an equivalent problem (G', k'), where $k' \leq k$ (excluding the trivial case). Then they apply the algorithm of Nemhauser and Trotter to $G' = (V', E')$. This requires $O(k^3)$ time, since G' contains $O(k^2)$ nodes and edges. Let $G'' = G[A(V')] = (V'', E'')$ and $k'' = k' - |B(V')| \leq k'$. Since $mvc(G'') \geq |V''|/2$, if the order of G'' is greater than $2k''$, it follows that $mvc(G'') > k''$, and thus $mvc(G') > k'$. Otherwise, the problem (G, k) is equivalent to (G'', k''), where $k'' \leq k$ and G'' contains at most $2k''$ nodes.

Proposition 1.2.3 *Let (G, k) be an instance of vertex cover, where the order of G is n. The algorithm above, of running time $O(kn + k^3)$, solves (G, k) or reduces it to an equivalent problem (G'', k''), where $k'' \leq k$ and G'' has at most $2k''$ nodes.*

Figure 11 A $O(2^k n)$ algorithm for vertex cover.

```
1    int vc1(G, k) {
2          remove nodes of degree 0 from G;
3          if(G contains no edge) return 0;
4          if(k = 0) return +∞;
5          take an edge {u, v} of G;
6          m1 = vc(G[V\{u}], k − 1);
7          m2 = vc(G[V\{v}], k − 1);
8          return threshold(min{1 + m1, 1 + m2}, k);
9    }
```

```
1    int threshold(p, k) {
2          if(p ≤ k) return p;
3          else return +∞;
4    }
```

1.2.2 Bounded Search Trees

Let us consider for simplicity a slightly different version of vertex cover problem, where the answer to an instance (G, k) is *yes* if and only if G has a vertex cover of at most k nodes (this version differs from the original one only in the trivial case, that is when the order of G is smaller than k). For simplicity, let us define a function $vc(\cdot, \cdot)$ as follows. For a given instance (G, k) of vertex cover:

$$vc(G, k) = \begin{cases} mvc(G) & \text{if } mvc(G) \leq k; \\ +\infty & \text{otherwise.} \end{cases}$$

The value $+\infty$ is simply used to easily take care of the instances for which a vertex cover of the desired size does not exist. Clearly, an algorithm which computes $vc(\cdot, \cdot)$, also solves vertex cover.

To introduce the idea behind bounded search trees, we consider the $O(2^k n)$ algorithm of Fellows [33]. The algorithm, which we denote by `vc1`, is shown in Figure 11. Algorithm `vc1` is based on the following simple observation. For any edge $\{u, v\}$ of G, every vertex cover contains u or v (or both). Thus one can branch by including u or v in the vertex cover. When the algorithm includes a node in the vertex cover, it removes it (and all the edges incident on it) from the graph, and it decreases the parameter k by one. In more details,

let (G, k) be the input instance of the problem. The algorithm works as follows. Nodes of degree zero are filtered out. For any k, if G contains no edge, the answer is 0. Otherwise, if $k = 0$ the answer is $+\infty$. Otherwise the algorithm takes an arbitrary edge $\{u, v\}$, and branches by including v or u in the vertex cover.

Proposition 1.2.4 *Algorithm* `vc1` *solves an instance* (G, k) *of vertex cover in* $O(2^k n)$ *time, where* n *is the number of nodes of* G.

Proof. The correctness of the algorithm easily follows from Lemma 1.2.1 and from the definition of vertex cover.

Let $N_h(k)$ denote the number of subproblems with parameter h created to solve a problem with parameter k. Clearly, $N_k(k) = 1$ and $N_h(k) = 0$ for $h > k$. Consider now the case $h < k$. For the trivial instances, $N_h(k) = 0$. Otherwise the algorithm branches on two subproblems $(G', k-1)$ and $(G'', k-1)$. Thus:

$$N_h(k) \leq 2N_h(k-1).$$

A valid upper bound for $N_h(k)$ is $N_h(k) \leq 2^{k-h}$. It follows that the total number $N(k)$ of subproblems generated is

$$N(k) = \sum_{h=0}^{k} N_h(k) \leq \sum_{h=0}^{k} 2^{k-h} = O(2^k).$$

The cost of solving a problem, excluding the cost of solving the corresponding subproblems (if any), is $O(n)$. Thus the time complexity of the algorithm is $O(2^k n)$. □

We will now describe another simple algorithm for vertex cover, which we denote by `vc2`, which generates a smaller search tree. The algorithm is described in Figure 12. The initial part of `vc2` coincides with the one of `vc1`. Then, if there is a node v of degree one, $N(v) = \{w\}$, `vc2` includes w in the vertex cover. Otherwise (all the nodes have degree greater than or equal to two), the algorithm takes a node v and branches by including v or $N(v)$ in the vertex cover.

Figure 12 A $O(1.62^k n^2)$ algorithm for vertex cover.

```
1   int vc2(G, k) {
2       remove nodes of degree 0 from G;
3       if(G contains no edge) return 0;
4       if(k = 0) return +∞;
5       if(there is a node v: N(v) = {w}) {
6           m=vc(G[V\{w}], k − 1);
7           return threshold(1 + m, k);
8       }
9       take a node v;
10      m1=vc(G[V\{v}], k − 1);
11      m2=vc(G[V\N(v)], k − |N(v)|);
12      return threshold(min{1 + m1, |N(v)| + m2}, k);
13  }
```

Proposition 1.2.5 *Algorithm* `vc2` *solves an instance* (G, k) *of vertex cover in* $O(1.62^k n^2)$ *time, where* n *is the number of nodes of* G.

Proof. The correctness of the algorithm follows from Lemma 1.2.1.

Let $N_h(k)$ denote the number of subproblems with parameter h created to solve a problem with parameter k. Again one has $N_k(k) = 1$ and $N_h(k) = 0$ for $h > k$. Consider now the case $h < k$. For the trivial instances, $N_h(k) = 0$. If there is a node of degree one, the algorithm generates a unique subproblem (G', k'), where $k' < k$. Then:

$$N_h(k) \leq N_h(k-1).$$

Otherwise, the algorithm generates two subproblems $(G', k-1)$ and (G'', k''), where $k'' \leq k-2$. Thus:

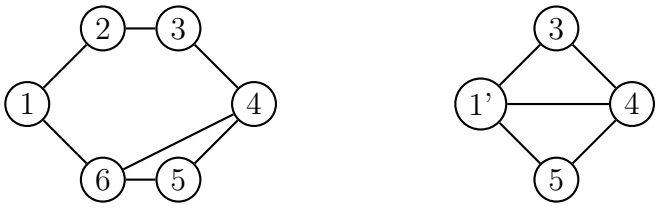
$$N_h(k) \leq N_h(k-1) + N_h(k-2).$$

A valid upper bound for $N_h(k)$ is $N_h(k) \leq c^{k-h}$ where $c = 1.61\ldots < 1.62$ is the (unique) positive root of the polynomial $(x^k - x^{k-1} - x^{k-2})$. It follows that the total number $N(k)$ of subproblems generated is $O(c^k)$. The cost of solving a problem, excluding the cost of solving the corresponding subproblems (if any), is $O(n^2)$. Thus the time complexity of the algorithm is $O(c^k n^2) = O(1.62^k n^2)$. □

Figure 13 A $O(1.47^k n^2)$ algorithm for vertex cover.

```
1    int vc3(G, k) {
2        remove nodes of degree 0 from G;
3        if(G contains no edge) return 0;
4        if(k = 0) return +∞;
5        if(there is a node v: N(v) = {w}) return threshold(1+vc(G[V\{w}], k − 1), k);
9        if(there is a node v: N(v) = {u, w}, u ≠ w) {
10           if(u and w are adjacent) return threshold(2+vc(G\{u, w}, k − 2), k);
11           else return threshold(1+vc(fold(G, v), k − 1), k);
12       }
13       take a node v;
14       m1=vc(G[V\{v}], k − 1);
15       m2=vc(G[V\N(v)], k − |N(v)|);
16       return threshold(min{1 + m1, |N(v)| + m2}, k);
17   }
```

Figure 14 The graph on the right is obtained by folding node 1 of the graph on the left.



Algorithms `vc1` and `vc2` generate subproblems which involve induced subgraphs of the original graph. This is not always the case. Algorithm `vc3`, which is described in Figure 13, generates subproblems involving graphs which are not induced subgraphs of the original graph. The importance of this distinction will be clearer in next section, where we will introduce a dynamic programming technique which applies only to the algorithms of the first kind (that is, like `vc1` and `vc2`). Algorithm `vc3` is based on the *vertex folding* technique of Chen et al. [14]. By *folding* a node v, we mean introducing a new node v', connecting v' with all the neighbors of the nodes in $N(v)$ and removing $N[v] = N(v) \cup \{v\}$ from the graph. We denote the new graph obtained by $fold(G, v)$. In Figure 14 an example of folding is depicted.

Let v be a node of degree two, $N(v) = \{u, w\}$. For Lemma 1.2.1, we know that there is

a minimum vertex cover which contains both u and w or neither of them. If u and w are adjacent, then one can impose that they are in every vertex cover. Otherwise, the problem (G, k) considered is equivalent to the problem $(fold(G, v), k - 1)$.

Proposition 1.2.6 *Let G be an undirected graph. Let moreover v be a node of G such that $N(v) = \{u, w\}$, where u and w are not adjacent. Then:*

$$mvc(G) = 1 + mvc(fold(G, v)).$$

Proof. Let V' be a minimum vertex cover of G. The set V' either contains both u and w or neither of them. In the first case, $V'' = V' \backslash \{u, w\} \cup \{v'\}$ is a vertex cover of $fold(G, v)$, where v' is the new node introduced with folding. In the second case, $V'' = V' \backslash \{v\}$ is a vertex cover of $fold(G, v)$. Thus:

$$mvc(G) \geq 1 + mvc(fold(G, v)).$$

Let now V'' be a minimum vertex cover of $fold(G, v)$. If v' belongs to V'', then $V' = V'' \backslash \{v'\} \cup \{u, w\}$ is a vertex cover of G. Otherwise, $V' = V'' \cup \{v\}$ is a vertex cover of G. Thus:

$$mvc(fold(G, v)) \geq mvc(G) - 1.$$

The claim follows. □

It follows that:

Corollary 1.2.2 *Let (G, k) be an instance of vertex cover. Let moreover v be a node of G such that $N(v) = \{u, w\}$, where u and w are not adjacent. Then:*

$$vc(G, k) = 1 + vc(fold(G, v), k - 1).$$

Proof. If $mvc(G) = m = 1 + mvc(fold(G, v))$, with $m \leq k$, then:

$$vc(G, k) = m = 1 + m - 1 = 1 + vc(fold(G, v), k - 1).$$

Otherwise:

$$vc(G, k) = +\infty = 1 + \infty = 1 + vc(fold(G, v), k - 1).$$

□

The initial part of `vc3` coincides with the one of `vc2`. Then, if there is a node v of degree two, the algorithm either includes $N(v) = \{u, w\}$ in the vertex cover (if u and v are adjacent), or folds node v. Otherwise, it takes an arbitrary node v and branches by including v or $N(v)$ in the vertex cover.

Proposition 1.2.7 *Algorithm* `vc3` *solves an instance* (G, k) *of vertex cover in* $O(1.47^k n^2)$ *time, where* n *is the number of nodes of* G.

Proof. The correctness of the algorithm follows from Lemma 1.2.1 and Corollary 1.2.2.

As usual, by $N_h(k)$ we denote the number of subproblems with parameter h created to solve a problem with parameter k. Again, $N_k(k) = 1$ and $N_h(k) = 0$ for $h > k$. Consider the case $h < k$. For the trivial instances, $N_h(k) = 0$. If there is a node of degree one or two, the algorithm generates a unique subproblem (G', k'), where $k' < k$. Then:

$$N_h(k) \leq N_h(k-1).$$

Otherwise, the algorithm generates two subproblems $(G', k-1)$ and (G'', k''), where $k'' \leq k-3$. Thus:

$$N_h(k) \leq N_h(k-1) + N_h(k-3).$$

A valid upper bound for $N_h(k)$ is $N_h(k) \leq c^{k-h}$ where $c = 1.46\ldots < 1.47$ is the (unique) positive root of the polynomial $(x^k - x^{k-1} - x^{k-3})$. This implies that the total number of problems generated is $O(1.47^k)$. The cost associated to each node is $O(n^2)$ (that is, the size of graph). Thus the time complexity of the algorithm is $O(1.47^k n^2)$. □

We are now ready to abstract the structure of an algorithm based on bounded search trees. Given a (non-trivial) problem (G, k), the algorithm generates b subproblems (G_1, k_1), (G_2, k_2)... (G_b, k_b), where b is bounded by a constant and $k_i < k$ for every i, and the algorithm solves them recursively. The number and kind of subproblems generated varies

depending on both the algorithm and the instance considered. Let $m_i = vc(G_i, k_i)$ and:

$$m = \min\{k - k_1 + m_1, k - k_2 + m_2 \dots k - k_b + m_b\}.$$

The subproblems are chosen in such a way that the following property holds:

$$vc(G, k) = \begin{cases} m & \text{if } m \leq k; \\ +\infty & \text{otherwise.} \end{cases}$$

Thus the solution of the subproblems directly leads to the solution of the original problem.

Let $N_h(k)$ denote the number of subproblems with parameter h created to solve a problem with parameter k. A valid upper bound on $N_h(k)$, for $h < k$, can be found in the following way. Consider a problem (G, k) with the corresponding subproblems (G_1, k_1), $(G_2, k_2) \dots (G_b, k_b)$. The following relation holds

$$N_h(k) \leq N_h(k_1) + N_h(k_2) \dots N_h(k_b).$$

The inequality above is satisfied by $N_h(k) \leq \tilde{c}^{k-h}$ where $\tilde{c}$, the *branching factor* associated to (G, k), is the (unique [14]) positive root of the polynomial $(x^k - x^{k_1} - x^{k_2} \dots - x^{k_b})$. Note that the value of $\tilde{c}$ only depends on the *branching vector* $(h_1, h_2 \dots h_b) = (k - k_1, k - k_2 \dots k - k_b)$. The *branching factor* c of the algorithm is the maximum over all the branching factors associated to the branchings that the algorithm may execute. By induction, one obtains that $N_h(k)$ is $\mathrm{O}(c^{k-h})$. It follows that the total number $N(k)$ of subproblems generated is $\mathrm{O}(c^k)$. The idea behind bounded search trees technique is to exploit the structure of the graph such as to obtain a branching factor c as small as possible. For this purpose, a tedious case analysis is often required.

Note that, by combining a $\mathrm{O}(c^k n^2)$ algorithm for vertex cover with the kernel reduction technique described in previous section, one obtains a $\mathrm{O}(c^k k^2 + kn)$ algorithm for the same problem. This complexity can be reduced to $\mathrm{O}(c^k + kn)$ via the *interleaving technique* of Niedermeier and Rossmanith [63].

1.2.3 Dynamic Programming

Memorisation is a dynamic programming technique developed by Robson [72, 73] in the context of maximum independent set, which allows to reduce the time complexity of many exponential-time recursive algorithms at the cost of an exponential space complexity. In this section we describe how Niedermeier and Rossmanith applied memorisation to vertex cover [65, 66]. In Chapter 2 we will present a more efficient way of doing that.

Let $\mathcal{A}$ be an algorithm of the kind described in previous section, with the extra condition that, when it branches at a problem (F, h), the corresponding subproblems (F_i, h_i) involve graphs F_i which are induced subgraphs of G. In particular, let us consider the fastest algorithm of this kind, which is the $O(c^k k^2 + kn) = O(1.2918^k k^2 + kn)$ algorithm of Niedermeier and Rossmanith [62]. From $\mathcal{A}$ one derives a new algorithm $\mathcal{A}'$ in the following way. For any induced subgraph F of the kernel of at most $2k' = 2\lfloor \alpha k \rfloor$ nodes, for a given $\alpha \in [0, 1]$, $\mathcal{A}'$ solves (by using $\mathcal{A}$) the problem (F, k') and it stores the pair $(F, vc(F, k'))$ in a database. Note that, since F is an induced subgraph of the kernel, one can simply store the set of nodes of F instead of F. Then $\mathcal{A}'$ works in the same way as $\mathcal{A}$, with the following difference. When the parameter h in a subproblem (F, h) reaches or drops below the value k', $\mathcal{A}'$ performs a kernel reduction. This way, $\mathcal{A}'$ generates a new subproblem (F', h'), where $h' \leq k'$ and the order of F is at most $2k'$. Thus $\mathcal{A}'$ can easily derive the value of $vc(F, h)$ from the value of $vc(F, k')$ which is stored in the database.

The cost of solving each subproblem (F, k') is $O(c^{k'}) = O(c^{\alpha k})$. The number of pairs stored in the database is at most $2\binom{2k}{2k'}$, which is $O(k^{-0.5}(\alpha^\alpha(1-\alpha)^{1-\alpha})^{-2k})$ from Stirling's approximation. Thus the database can be created in $O(c^{\alpha k} k^{-0.5}(\alpha^\alpha(1-\alpha)^{1-\alpha})^{-2k})$ time. The search tree now contains $O(c^{(1-\alpha)k})$ nodes (that is the upper bound on $N(k)$ which is obtained by assuming $N(k') = 1$). The cost associated to each node is $O(k^2)$, not considering the leaves for which the costs of the query to the database and of the kernel reduction have to be taken into account. In particular, the database can be implemented such as that

the cost of each query is $O(k)$. Moreover each kernel reduction costs $O(k^{2.5})$ (since each graph contains $O(k)$ nodes and $O(k^2)$ edges). Thus the cost to create the search tree is $O(c^{(1-\alpha)k}k^{2.5})$. The value of α has to be chosen such as to balance the cost of creating the search tree and the database. The optimum is reached when $\alpha > 0.0262$ satisfies:

$$c^{1-\alpha} = c^{\alpha}\left(\frac{1}{\alpha^{\alpha}(1-\alpha)^{1-\alpha}}\right)^2.$$

Thus one obtains a $O(c^{(1-\alpha)k}k^{2.5} + kn) = O(1.2832^k k^{2.5} + kn)$ time complexity.

The complexity can be slightly reduced in the following way. Consider an algorithm of complexity $O(c^k k^{2.5} + kn)$. All the nodes of degree greater than a given Δ are filtered out in a *preliminary phase* (in which the algorithm stores no solution in the database and no kernel reduction is performed). In particular, let (F, h) be a subproblem where F contains a node v with $deg(v) > \Delta$. The algorithm branches on the subproblems $(F[V\backslash\{v\}], h-1)$ and $(F[V\backslash N(v)], h - |N(v)|)$. The number of subproblem generated in the preliminary phase is $O(\tilde{c}^k)$, where $\tilde{c}$ is the positive root of the polynomial $(x^{\Delta+1} - x^{\Delta} - 1)$. The cost of solving these subproblems, excluding the cost of solving the corresponding subproblems (if any), is $O(k^2)$. Thus the total cost to remove "high" degree nodes is $O(\tilde{c}^k k^2)$. All the subproblems generated after the preliminary phase, involve subgraphs with $O(k)$ nodes and edges. This means that the kernel reductions can be performed in $O(k^{1.5})$ time only [65]. For a big enough Δ, one has $\tilde{c} < c$. Thus the complexity of the algorithm can be reduced to $O(\tilde{c}^k k^2 + c^k k^{1.5} + kn) = O(c^k k^{1.5} + kn)$. In particular, by imposing $\Delta = 6$, one has $\tilde{c} = 1.255\ldots < 1.256$. Thus the complexity of the algorithm of Niedermeier and Rossmanith is reduced to $O(1.2832^k k^{1.5} + kn)$.

1.3 Cliques

A *clique* is a undirected graph such that its nodes are pairwise adjacent. A *k-clique* is a clique of order k. The 3-cliques are also called *triangles*. The *clique* problem consists in

determining whether an undirected graph $G = (V, E)$ with n nodes contains a clique of order k, that is whether there is a subset V' of k nodes such that $G[V']$ is a clique.

Itai and Rodeh [44] provided a $O(n^\omega)$ algorithm for clique problem in the case $k = 3$. The algorithm works as follows. It computes A^3 via a fast square matrix multiplication subroutine, where A is the adjacency matrix of G. The answer is *yes* if and only if A^3 contains a non-zero element in the main diagonal.

Proposition 1.3.1 *The algorithm above determines whether a graph G of n nodes contains a triangle in $O(n^\omega)$ time.*

Proof. Remember that, for every node i, $A[i, i] = 0$. The number of paths of length three between two nodes i and j is $A^3[i, j]/2$. In particular, the number of triangles in which a node i is contained is $A^3[i, i]/2$. Thus G contains a triangle if and only if A^3 has a non-zero entry in the main diagonal.

The two multiplications needed to compute A^3 can be performed in $O(n^\omega)$ time each. □

Nešetřil and Poljak [61] generalized the idea of Itai and Rodeh to the detection of cliques of arbitrary size. Their algorithm works as follows. If $k \pmod 3 \neq 0$, it recursively searchs for a $(k-1)$-clique in the graphs $G[N(v)]$, for every node v. The answer is *yes* if and only if at least one of the graphs $G[N(v)]$ contains a $(k-1)$-clique. If $k = 3h$, for some $h \in \mathbb{N}$, the algorithm creates an auxiliary graph $\widetilde{G}$ which has a node for each h-clique in G, and an edge between a pair of nodes if and only if the corresponding nodes in G form a $(2h)$-clique. The answer is *yes* if and only if $\widetilde{G}$ contains a triangle.

Proposition 1.3.2 *The algorithm above determines whether a graph G of n nodes contains a clique of k nodes in $O(n^{\alpha(k)})$ time, where $\alpha(k) = \omega\lfloor k/3 \rfloor + k \pmod 3$.*

Proof. Clearly, a node v is contained in a k-clique, for every $k \geq 2$, if and only if the graph $G[N(v)]$ induced by its neighborhood contains a $(k-1)$-clique. Consider now the case that $k = 3h$, $h \in \mathbb{N}$. Then G contains a k-clique if and only if $\widetilde{G}$ contains a triangle. In fact, assume that G contains a k-clique $\{v_1, v_2 \ldots v_k\}$. Thus $\widetilde{G}$ contains a node for each one of the

three cliques $\{v_1, v_2 \dots v_h\}$, $\{v_{h+1}, v_{h+2} \dots v_{2h}\}$, $\{v_{2h+1}, v_{2h+2} \dots v_k\}$. Moreover these nodes are pairwise adjacent. Thus $\widetilde{G}$ contains a triangle.

Assume now that $\widetilde{G}$ contains a triangle $\{w_1, w_2, w_3\}$. Note that w_i and w_j, $i \neq j$, cannot contain the same node v of G, since otherwise their nodes could not form a $2h$-clique in G. Let $T = \bigcup_i w_i$. Every two distinct nodes of T are adjacent in G. Thus T is a subset of $3h = k$ pairwise adjacent nodes of G, that is a k-clique of G.

Consider the time complexity of the algorithm. The number of auxiliary graphs created by the algorithm is $O(n^{k \pmod 3})$. An auxiliary graph can be created in $O(n^{2\lfloor k/3 \rfloor})$ time. Since each auxiliary graph contains $O(n^{\lfloor k/3 \rfloor})$ nodes, a triangle in it can be detected in $O(n^{\omega \lfloor k/3 \rfloor})$ time, via the algorithm of Itai and Rodeh. Thus the complexity of the algorithm is $O(n^{\omega \lfloor k/3 \rfloor + k \pmod 3})$ □

Alon, Yuster and Zwick [2] developed a fast algorithm to detect triangles in sparse graphs. Their approach was generalized to the detection of k-clique, for $k \geq 3$, by Kloks et al. [49]. The idea is to partition the vertex set into the set L of the nodes of degree smaller than $\Delta = e^{(\alpha(k)-1)/(\alpha(k)+\alpha(k-1)-1)}$ (*low degree nodes*), and the set H of the remaining nodes (*high degree nodes*). First one searches for a k-clique which contains at least one low degree node. Then one searches for a k-clique which contains high degree nodes only. In more details, for every node $v \in L$, the algorithm first searches for a $(k-1)$-clique in the graph $G[N(v)]$. If no k-clique is detected in this phase, the algorithm searches for a k-clique of the graph $G[H]$ induced by the high degree nodes. In both phases, the cliques are searched for with an arbitrary algorithm for clique detection in dense graphs. In particular, the algorithm of Nešetřil and Poljak can be used.

Proposition 1.3.3 *The algorithm above determines whether a graph G with e edges contains a k-clique in $O(e^{\alpha(k)\alpha(k-1)/(\alpha(k)+\alpha(k-1)-1)})$ time.*

Proof. The correctness of the algorithm is trivial.

A k-clique containing at least one low degree node can be detected in $O(\sum_{v \in L} deg(v)^{\alpha(k-1)})$

$= O(e\Delta^{\alpha(k-1)-1})$ steps. Since $\Sigma_{v \in V}\ deg(v) = 2\,e$, the number of high degree nodes is bounded by $|H| \leq 2\,e/\Delta$. Then a clique formed by high degree nodes only can be detected in $O\left((e/\Delta)^{\alpha(k)}\right)$ time. The complexity of the procedure is then $O(e^{\alpha(k)\alpha(k-1)/(\alpha(k)+\alpha(k-1)-1)})$.

□

The (Induced) Subgraph Problem

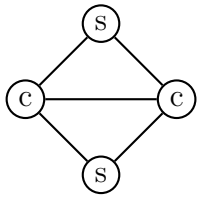
Nešetřil and Poljak [61] showed how to reduce the detection of an arbitrary (induced) subgraph of fixed order, to the detection of a clique of the same order in an auxiliary graph. In more details, consider the detection of an induced subgraph F of fixed order k. The graph G contains such an induced subgraph if and only if a clique of order k is contained in the following auxiliary graph $\widetilde{G}$. Graph $\widetilde{G}$ has a node (v, w), for each pair of nodes v of F and w of G. Two nodes (v_1, w_1) and (v_2, w_2) are adjacent if $v_1 \neq v_2$, $w_1 \neq w_2$ and v_1 is adjacent to v_2 if and only if w_1 is adjacent to w_2. Graph $\widetilde{G}$ contains $k\,n$ nodes. Thus, for every fixed k, a k-clique of $\widetilde{G}$ can be detected in $O(n^{\alpha(k)})$ steps. The detection of a subgraph is analogous. The main difference is that now two nodes (v_1, w_1) and (v_2, w_2) are adjacent if $v_1 \neq v_2$, $w_1 \neq w_2$ and if v_1 and v_2 are adjacent, then also w_1 and w_2 are adjacent (the viceversa is not required any more).

Corollary 1.3.1 *The algorithm above determines whether an undirected graph G with n nodes contains an (induces) subgraph F of k nodes in $O(n^{\alpha(k)})$ time, for every fixed $k \geq 3$.*

1.4 Diamonds

A *diamond* is obtained from a 4-clique by removing one edge. In a diamond there are two nodes of degree three and two of degree two. We call *central* nodes the nodes of the first kind, and *side* nodes the other two nodes. A *diamond* is represented in Figure 15. A graph is *diamond-free* if it does not contain an induced diamond. Detecting diamonds is important since maximum clique is solvable in polynomial time in diamond-free graphs. In particular, the following results hold [78].

Figure 15 A diamond. Side and central nodes are labelled with s and c respectively.



Lemma 1.4.1 *A graph G is diamond-free if and only if, for every node v, the graph $G[N(v)]$ induced by the neighborhood of v is a disjoint union of cliques.*

Corollary 1.4.1 *Maximum clique on a diamond-free graph of n nodes and e edges is solvable in $\mathrm{O}(n(n+e))$ time.*

Kloks et al. [49] provided a $\mathrm{O}(n^{\omega}+e^{3/2})$ algorithm for diamond detection. Their algorithm works as follows. The nodes are partitioned in the set H of nodes of degree greater than $\Delta = \sqrt{e}$ (*high degree* nodes), and the set L of the remaining nodes (*low degree* nodes). Then the algorithm executes the following steps:

1. It searchs for a diamond with a central node in L. In particular, for every $v \in L$, it computes the connected components of $G[N(v)]$ and it checks whether they are all cliques. If not, for Lemma 1.4.1, v is contained in a diamond.

2. If no diamond is detected in the previous step, the algorithm looks for a diamond with a side node in L. In particular, it computes A^2 via fast matrix multiplication, where A is the adjacency matrix of G. Consider a node $v \in L$. From the previous step, the connected components of $G[N(v)]$, which must be cliques, are already known. For every clique K of $G[N(v)]$ of order p and for every pair of node w and u of K, the algorithm checks whether $A^2[u,w] > p-1$. In that case, u and w belong to a diamond. In fact, v, u and w form a triangle, and there must be a node x not in $N(v)$ which is adjacent to both u and w.

3. If no diamond is found in the previous step, the algorithm looks for a diamond formed by high degree nodes only. In particular, it applies the same approach of the first step to $G[H]$.

Proposition 1.4.1 *The algorithm above determines whether a graph of n nodes and e edges contains an induced diamond in $O(n^\omega + e^{3/2})$ time.*

Proof. The correctness of the algorithm follows from Lemma 1.4.1 and from simple considerations.

The first step costs $O(\sum_{v \in L} deg(v)^2) = O(e\Delta)$. The second step costs $O(n^\omega + \sum_{v \in L} deg(v)^2) = O(n^\omega + e\Delta)$. The number of high degree nodes is bounded by $|H| \leq 2e/\Delta$. Thus the third step costs $O(\sum_{v \in H} deg(v)^2) = O(e^2/\Delta)$. It follows that the time complexity of the algorithm is $O(n^\omega + e^{3/2})$. □

1.5 Cycles

There is a simple $O(n^\omega)$ algorithm which allows to determine whether a graph G, both directed or undirected, contains a k-cycle, for a fixed $k \geq 3$. In fact, consider the matrix A^k, where A is the adjacency matrix of G. Such matrix can be computed with $O(1)$ matrix multiplications, each one of cost $O(n^\omega)$. A node v belongs to a cycle of length k if and only if $A^k[v, v] > 0$.

Proposition 1.5.1 *The algorithm above determines whether a graph G, both directed or undirected, of n nodes contains a subgraph isomorphic to a k-cycle in $O(n^\omega)$ time.*

Note that the algorithm above does not allow to distinguish between simple and non-simple cycles. Alon, Yuster and Zwick [1] introduced the method of *color coding*, which leads to a randomized algorithm to detect a C_k, a simple k-cycle, in a graph which contains at least one C_k. Consider the directed case first. The algorithm works as follows. Until a C_k is detected, the following steps are repeated. The algorithm randomly colors G with k colors $1, 2, \ldots k$. Let $c(v)$ be the color of node v. Then it creates an auxiliary graph $\widetilde{G}$ which is obtained by

removing from G all the edges (u, v) which do not satisfy the property: $c(u) = (c(v) + 1) \pmod{k}$. Eventually the algorithm searches for a k-cycle of $\widetilde{G}$, in $O(n^\omega)$ time. Note that the k-cycles of $\widetilde{G}$ are simple, since they are formed by nodes of k distinct colors. Moreover any such cycle corresponds to a C_k of G. In particular, $\widetilde{G}$ contains a given C_k of G with a finite positive (though "small") probability. Thus the algorithm halts in a constant expected number of rounds for every fixed k. The undirected case can be easily reduced to the directed case, by replacing each undirected edge $\{u, v\}$ with the two directed edges (u, v) and (v, u) (for each undirected C_k there are two directed C_k). Note that the algorithm above does not halt if G dos not contain a simple k-cycle.

Alon et al. [1] showed how to derandomize the color coding technique, with a logarithmic increase in the time complexity. The idea is to replace the random colorings with a (as small as possible) family of colorings such that every subset of k nodes is assigned distinct colors in at least one of these colorings. In other words, one needs a family of *perfect hash function* from $\{1, 2 \dots n\}$ to $\{1, 2 \dots k\}$. Schmidt and Siegel [74] described a family of this kind formed by $2^{O(1)} \log^2 n$ functions. Alon et al. [1] showed how to obtain a family of $k^{O(k)} \log n$ functions. Note that, for every fixed k, the size of this family is $O(\log n)$.

Proposition 1.5.2 *The algorithm above determines whether a graph G, both directed or undirected, of n nodes contains a subgraph isomorphic to a C_k in $O(n^\omega \log n)$ time.*

Monien [57] described a $O(e)$ algorithm to detect a C_k, for fixed $k \geq 3$, which passes through a given node v. This implies that a C_k can be detected in $O(en)$ time. Alon, Yuster and Zwick [2] used this result to develop a $O(e^{2-1/k})$ algorithm to detect a C_{2k-1} and a C_{2k}. Consider the detection of a C_{2k} (the other case is analogous). The idea is again to partition the set of nodes into the nodes H of degree greater that $\Delta = e^{1/k}$ and the set L of the remaining nodes. Since $|H| \leq 2e/\Delta$, the simple k-cycles which contain at least one node in H can be detected in $O(e^2/\Delta)$ time. There are at most $O(e\Delta^{k-1})$ simple paths of length k (which can be enumerated in $O(e\Delta^{k-1})$ time). Once that such paths are available, a C_{2k}

formed by nodes in L only can be detected in linear time in the number of paths using the technique described in [2]. Thus the complexity of the algorithm is $O(e^{2-1/k})$.

Proposition 1.5.3 *The algorithm above determines whether a graph G, both directed or undirected, of e edges contains a subgraph isomorphic to a C_k in $O(e^{2-1/\lceil k/2 \rceil})$ time.*

Faster algorithms exists to detect even cycle in undirected graphs. Some of these results rely on the fact that dense graphs contain "many" even cycles. In particular, the following lemma of Bondy and Simonovits holds [10].

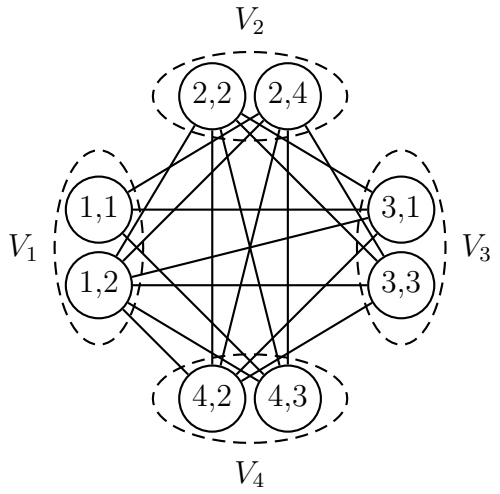
Lemma 1.5.1 *Let G be an undirected graph of n nodes and e edges. If $e > 100kn^{1+1/k}$ for some positive integer k, then G contains a C_{2h} for every integer $h \in [k, n^{1/k}]$.*

1.6 The Binary Constraints Satisfaction Problem

A *binary constraints network* $\mathcal{N}$ is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{1, 2 \dots n\}$ is a set of n variables, $\mathcal{D} = \{D_1, D_2 \dots D_n\}$ is a set of n domains of cardinality at most d, and $\mathcal{C}$ is a set of e binary constraints. Variable i is defined over domain D_i. A *value assignment* is a pair (i, a), where $a \in D_i$, whose meaning is that we assign the value a to variable i. An *instantiation* is a set of value assignments, exactly one for each variable. A *binary constraint* $C_{\{i,j\}}$ describes which assignments of values to the variables i and j are mutually compatible. The constraint $C_{\{i,j\}}$ can be represented extensively through a 0-1 matrix $A_{i,j}$ which we interpret in the following way: $A_{i,j}[a, b] = 1$ if and only if the pair of value assignments (i, a) and (j, b) satisfies the constraint $C_{\{i,j\}}$. Notice that there may be pairs of variables i and j for which there is no constraint $C_{\{i,j\}}$ in $\mathcal{C}$. In that case all the assignments of values to i and j are implicitly assumed to be mutually compatible. Two variables i and j are *linked* if there is a constraint $C_{\{i,j\}}$ in $\mathcal{C}$. A *solution* is an instantiation which satisfies all the constraints. The *binary constraint satisfaction problem* consists in determining whether a binary constraints network admits a solution.

The binary constraints network $\mathcal{N}$ can be represented through a n-partite graph $G = (\{V_1, V_2 \dots V_n\}, E)$, which we call *consistency graph*. The consistency graph has a partition

Figure 16 Example of binary constraints network with the corresponding consistency graph. The nodes relative to the same variable are included into dashed ellipses.



$\mathcal{X} = \{x_1, x_2, x_3, x_4\}$,

$\mathcal{D} = \{D_1, D_2, D_3, D_4\}$,

$D_1 = \{1, 2\}$,$D_2 = \{2, 4\}$,$D_3 = \{1, 3\}$,$D_4 = \{2, 3\}$,

$\mathcal{C} = \{C_{\{1,2\}}, C_{\{1,3\}}, C_{\{1,4\}}, C_{\{2,3\}}, C_{\{3,4\}}\}$,

$A_{1,2}[a, b] = 0$ iff $(a, b) \in \{(1, 2)\}$,
$A_{1,3}[a, b] = 0$ iff $(a, b) \in \{(1, 3)\}$,
$A_{1,4}[a, b] = 0$ iff $(a, b) \in \{(1, 2)\}$,
$A_{2,3}[a, b] = 0$ iff $(a, b) \in \{(4, 1)\}$,
$A_{3,4}[a, b] = 0$ iff $(a, b) \in \{(1, 3), (3, 3)\}$.

V_i for each variable i, which has a node for each element of D_i. Each node is labelled with the corresponding value assignment (i, a). Two nodes (i, a) and (j, b) are adjacent if $i \neq j$ and there is no constraint $C_{\{i,j\}} \in \mathcal{C}$ such that $A_{i,j}[a, b] = 0$. Notice that, if two variables are not linked, the corresponding partitions induce a complete bipartite graph on G. It is not hard to show that a *n-clique* in G corresponds to each solution of $\mathcal{N}$. Then the binary constraint satisfaction problem can be reduced to the problem of determining whether G contains a n-clique. In Figure 16 an example of binary constraints network with the corresponding consistency graph is represented. There are four variables, 1, 2, 3 and 4. In the example, all the domains have the same cardinality: this is not always the case. All the pairs of variables are linked, excluding the pair $\{2, 4\}$. Notice that Partitions V_2 and V_4 induce a complete bipartite graph. A solution is the instantiation $\{(1, 2), (2, 4), (3, 3), (4, 2)\}$.

From now on, we will refer to the consistency graph representation of the network. A node (i, a) is *consistent* if it belongs to a n-clique, and *inconsistent* otherwise. Inconsistent nodes can be removed from the graph without loosing any n-clique. Let $\mathcal{P}$ be a property which all the nodes in an n-clique need to satisfy. We denote by $\mathcal{G}_{\mathcal{P}}$ the set of the induced

subgraphs of G such that all their partitions are non-empty and all their nodes satisfy $\mathcal{P}$. If $\mathcal{G}_{\mathcal{P}}$ is empty, G cannot contain a n-clique. Otherwise, all the n-cliques of G are contained in the (unique) element $G_{\mathcal{P}}$ of $\mathcal{G}_{\mathcal{P}}$ of maximum order. A $\mathcal{P}$-based *filtering algorithm* is an algorithm which computes $G_{\mathcal{P}}$ or determines that $\mathcal{G}_{\mathcal{P}}$ is empty.

A node (i, a) is *arc consistent* if, for any $j \neq i$, there is a node (j, b) adjacent to (i, a). For example node $(4, 2)$ of Figure 16 is arc consistent (it is adjacent to $(1, 2)$, $(2, 2)$ and $(3, 1)$), while node $(4, 3)$ is not (it is not adjacent to any node in V_3). The idea behind arc consistency can be easily generalized. A node (i, a) is *path inverse consistent* [30] if it is arc consistent and, for any other two variables j and k, $i \neq j \neq k \neq i$, there are two nodes (j, b) and (k, c) which form a triangle with (i, a). For example node $(1, 2)$ of Figure 16 is path inverse consistent (triangles $\{(1, 2), (2, 2), (3, 1)\}$, $\{(1, 2), (2, 2), (4, 2)\}$ and $\{(1, 2), (3, 1), (4, 2)\}$), while node $(1, 1)$ is not (it is not contained in any triangle of the kind $\{(1, 1), (2, a), (4, b)\}$). The *$\ell$-inverse consistency* [30] is the natural generalization of arc consistency ($\ell = 2$) and path inverse consistency ($\ell = 3$) to an arbitrary (fixed) value of ℓ.

A node (i, a) is *max-restricted path consistent* if it has a path-consistent support on each variable j linked to i. A *path-consistent support* for (i, a) on variable j is a node (j, b), adjacent to (i, a), such that the pair $\{(i, a), (j, b)\}$ has at least one witness on each variable k linked to both i and j. A *witness* for $\{(i, a), (j, b)\}$ on k is a node (k, c) which forms a triangle with (i, a) and (j, b). Consider for example node $(1, 1)$ of Figure 16. We want to check whether $(1, 1)$ is max-restricted path consistent. Variable 2 is linked to variable 1. Then we need to search for a path-consistent support for $(1, 1)$ on 2. Node $(2, 2)$ cannot be a path-consistent support for $(1, 1)$ on 2 since it is not compatible with $(1, 1)$. Consider now node $(2, 4)$, which is compatible with $(1, 1)$. We have to search for a witness for the pair $\{(1, 1), (2, 4)\}$ on variable 3 only. In fact, variable 3 is linked to both variables 1 and 2, while variable 4 is not linked to variable 2. Both $(3, 1)$ and $(3, 3)$ are not witnesses for $\{(1, 1)(2, 4)\}$ on variable 3, since they are not adjacent to $(2, 4)$ and $(1, 1)$ respectively. Then

$(2,4)$ is not a path-consistent support for $(1,1)$. Since $(1,1)$ has no path-consistent support on variable 2, it is not max-restricted path consistent. Node $(1,2)$ instead, is max-restricted path consistent, since it is *supported* by $(2,4)$ on 2 (witness $(3,3)$), by $(3,3)$ on 3 (witnesses $(2,4)$ and $(4,2)$) and by $(4,2)$ on 4 (witness $(3,3)$).

Chapter 2

Vertex Cover

A *vertex cover* of an undirected graph $G = (V, E)$ of n nodes is a subset V' of nodes such that any edge is incident on at least one node in V'. The *vertex cover problem* consists in determining whether G has a vertex cover of k nodes. The currently fastest algorithm for vertex cover for small values of k is the $O(1.2832^k k^{1.5} + kn)$ exponential-space algorithm of Niedermeier and Rossmanith [65, 66]. Their algorithm makes use of *memorisation*, a dynamic programming technique proposed by Robson [72].

In this chapter we show how to obtain a $O(1.2745^k k^4 + kn)$ time complexity via a refined use of memorisation. The rest of this chapter is organized as follows. In Section 2.1, we present a variant of the exponential-space algorithm of Niedermeier and Rossmanith of complexity $O(1.2829^k k^{1.5} + kn)$. The reduction in the complexity is achieved via a more efficient use of the database. In Section 2.2, we show how to reduce the complexity to $O(1.2759^k k^{1.5} + kn)$ by branching on subproblems involving connected induced subgraphs only. In Section 2.3, the complexity is further reduced to $O(1.2745^k k^4 + kn)$.

2.1 A More Efficient Use of the Database

Our algorithm, which we denote by $\mathcal{A}$ works as the algorithm of Niedermeier and Rossmanith (see Section 1.2), with the following differences. No database is created a priori. When a subproblem (F, h) is solved the triple $(F, h, vc(F, h))$ is stored in a database (for every $h > 0$).

Before solving any subproblem (F, h), the algorithm checks whether the solution is already available in the database. This way one ensures that a given subproblem is solved at most once. Let (F, h) be a problem generated after the preliminary phase, in which nodes of degree higher than six are removed. Before branching on the corresponding subproblems $(F'_1, h'_1), (F'_2, h'_2) \ldots (F'_b, h'_b)$, as generated by the algorithm of Niedermeier and Rossmanith, $\mathcal{A}$ performs a kernel reduction. In particular, the algorithm branches on a set of subproblems $(F_1, h_1), (F_2, h_2) \ldots (F_b, h_b)$, where the order of F_i is at most $2h_i$ (*order-condition*). The reason of enforcing the order-condition will be clearer in the analysis. Observe that the kernel reduction does not increase the branching factor of the algorithm. Then $\mathcal{A}$ computes $vc(F, h)$ as usual. In particular, it computes $m_i = vc(F_i, h_i)$ for each $i = 1, 2 \ldots b$. The value of $vc(F, h)$ is given by:

$$vc(F, h) = \begin{cases} m & \text{if } m \leq h, \\ +\infty & \text{otherwise,} \end{cases}$$

where $m = \min_i\{h - h_i + m_i\}$.

Theorem 2.1.1 *Algorithm $\mathcal{A}$ solves vertex cover in $\mathrm{O}(1.2829^k k^{1.5} + kn)$ time, where n is the number of nodes in the graph and k is the size of the vertex cover.*

Proof. The correctness follows from the correctness of the algorithm described in Section 1.2.3.

The cost of the kernel reduction and of the preliminary phase is the same as in the algorithm of Niedermeier and Rossmanith, that is $\mathrm{O}(1.256^k k^2 + kn)$. The cost associated to each node of the search tree, after the preliminary phase, is $\mathrm{O}(k^{1.5})$.

Let $N_h(k)$ denote the number of subproblems with parameter h generated after the preliminary phase to solve a problem with parameter k. From the analysis of Section 1.2, $N_h(k)$ is $\mathrm{O}(c^{k-h})$ for every $h \leq k$, $c = 1.2917 \ldots < 1.2918$. The subproblems considered involve induced subgraphs of the kernel of order at most $2h$. Let $N(2k, 2h)$ denote the number of such subgraphs. Since no subproblem is solved more than once, $N_h(k)$ is $\mathrm{O}(\min\{c^{k-h}, N(2k, 2h)\})$. As h decreases from k to zero, the function $\min\{c^{k-h}, N(2k, 2h)\}$ first increases, then reaches

a peak, and eventually decreases. Note that, for $h \leq k/4$, $N(2k, 2h) \leq 2\binom{2k}{2h}$. Let $h' \leq k/4$ be the largest integer such that $2\binom{2k}{2h'} < c^{k-h'}$. The total number $N(k)$ of subproblems generated after the preliminary phase satisfies:

$$N(k) = \sum_{h=0}^{k} N_h(k) \leq \sum_{h=0}^{h'} 2\binom{2k}{2h} + \sum_{h=h'+1}^{k} c^{k-h} = \mathrm{O}(\binom{2k}{2h'} + c^{k-h'}) = \mathrm{O}(c^{k-h'}).$$

It follows from Stirling's approximation that $N(k)$ is $\mathrm{O}(c^{(1-\alpha)k})$, where α satisfies:

$$c^{1-\alpha} = \left(\frac{1}{\alpha^{\alpha}(1-\alpha)^{1-\alpha}}\right)^2.$$

The cost of solving a subproblem, excluding the cost of solving the corresponding subproblems (if any), is $\mathrm{O}(k^{1.5})$. Thus the complexity of the algorithm is $\mathrm{O}(c^{(1-\alpha)k}k^{1.5} + kn) = \mathrm{O}(1.2829^k k^{1.5} + kn)$. □

Besides the time complexity, an important difference between the algorithm proposed in this section and the exponential-space algorithm of Niedermeier and Rossmanith is the role played by the parameter α. In the algorithm of Niedermeier and Rossmanith, α influences the behavior of the algorithm, and thus it has to be fixed carefully. In our algorithm instead, α only appears in the complexity analysis.

2.2 Branching on Connected Induced Subgraphs

In previous section we described an algorithm $\mathcal{A}$ which makes use of a database in which it stores triples of the kind $(F, h, vc(F, h))$, where F is an induced subgraph of the kernel of order at most $2h$ (order-condition). The graphs F stored in the database may not be connected. We will now show how to derive from $\mathcal{A}$ an algorithm $\mathcal{A}_C$ with the same branching factor as $\mathcal{A}$, which branches only on subproblems involving induced subgraphs which are connected (besides satisfying the order-condition). As a consequence, $\mathcal{A}_C$ is asymptotically faster than $\mathcal{A}$.

The difference between $\mathcal{A}_C$ and $\mathcal{A}$ is the way $\mathcal{A}_C$ branches at a given subproblem after the preliminary phase. Let (F, h) be a problem generated after the preliminary phase and

$(F_1, h_1), (F_2, h_2) \dots (F_b, h_b)$ be the corresponding subproblems generated by $\mathcal{A}$. Consider the connected components $F_{i,1}, F_{i,2} \dots F_{i,p_i}$ of F_i, $i = 1, 2 \dots b$. A naive idea could be to branch on the subproblems $(F_{i,j}, h_i)$, for each $i \in \{1, 2 \dots b\}$ and for each $j \in \{1, 2 \dots p_i\}$ (the graphs $F_{i,j}$ are connected and satisfy the order-condition). In fact, it is not hard to show that:

$$m_i = \begin{cases} \sum_j m_{i,j} & \text{if } \sum_j m_{i,j} \leq h_i; \\ +\infty & \text{otherwise,} \end{cases}$$

where $m_{i,j} = vc(F_{i,j}, h_i)$, $j = 1, 2 \dots p_i$.

Though this approach is correct in principle, it may lead to a bad branching factor. The reason is that one may generate many more than b subproblems, without decreasing the parameter in each subproblem properly. To avoid this problem, one can use the following simple observation. The size of the minimum vertex covers of the connected components with less than $6\ell + 2$ nodes, for a fixed positive integer ℓ, can be computed in constant time (by brute force). Thus one does not need to branch on them. The size of the minimum vertex covers of the remaining connected components is lower bounded by ℓ (since they contain at least $6\ell + 1$ edges and their degree is upper bounded by six). This lower bound can be used to reduce the parameter in each subproblem.

In more details, for each connected component H of F_i, $i = 1, 2 \dots b$, with less than $6\ell + 2$ nodes, the algorithm computes $mvc(H)$ by brute force, and this value is added to a variable Δ_i (which is initialized to zero). Let $F_{i,1}, F_{i,2} \dots F_{i,b_i}$ be the remaining connected components of F_i. The minimum vertex covers of each $F_{i,j}$ have cardinality at least ℓ. This implies that, if the size of the minimum vertex covers of some $F_{i,j}$ is greater than $h'_i = h_i - (b_i - 1)\ell$, then $vc(F_i, h_i) = +\infty$. Thus one can compute $m'_{i,j} = vc(F_{i,j}, h'_i)$ instead of $m_{i,j} = vc(F_{i,j}, h_i)$. Note that the order-condition is satisfied by the new subproblems since each $F_{i,j}$ contains at most $2h_i - (b_i - 1)(6\ell + 2)$ nodes. It follows that

$$m_i = \begin{cases} \Delta_i + \sum_j m'_{i,j} & \text{if } \Delta_i + \sum_j m'_{i,j} \leq h_i; \\ +\infty & \text{otherwise.} \end{cases} \tag{2.1}$$

Once that the values m_i are available, the value of $vc(F, h)$ can be computed as usual. In

this case also, one does not need to compute the solutions of the subproblems $(F_{i,j}, h'_i)$ which are already available in the database.

We will now show how to choose ℓ such that the branching factor of $\mathcal{A}_C$ is not greater than the branching factor of $\mathcal{A}$. Consider a subproblem (F, h). Let $\hat{c}$ and $\hat{c}_C$ be the branching factors corresponding to the branching on (F, h) of the algorithms $\mathcal{A}$ and $\mathcal{A}_C$ respectively. We can decompose the branching of $\mathcal{A}_C$ on (F, h) in $b + 1$ branchings. First there is one branching on the subproblems $(F_1, h_1), (F_2, h_2) \ldots (F_b, h_b)$ (as generated by $\mathcal{A}$). Then, for each subproblem (F_i, h_i), $i \in \{1, 2 \ldots b\}$, there is one branching on the "big" connected components of F_i. The first branching has branching factor $\hat{c}$. The other branchings have branching factors $\hat{c}_1, \hat{c}_2 \ldots \hat{c}_b$ respectively, where:

$$\hat{c}_i = \begin{cases} 1 & \text{if } b_i = 1; \\ b_i^{\frac{1}{(b_i-1)\ell}} \leq 2^{1/\ell} & \text{otherwise.} \end{cases}$$

The value of $\hat{c}_C$ is upper bounded by the maximum over $\hat{c}$ and $\hat{c}_1, \hat{c}_2 \ldots \hat{c}_b$:

$$\hat{c}_C \leq \max\{\hat{c}, \hat{c}_1, \hat{c}_2 \ldots \hat{c}_b\}.$$

Thus, to ensure that the branching factor of $\mathcal{A}_C$ is not greater than the branching factor c of $\mathcal{A}$, it is sufficient to fix ℓ such that $2^{1/\ell} < c$. In particular, we can assume $\ell = 3$ in the case of the algorithm of Niedermeier and Rossmanith ($c > 1.2917 > 2^{1/3}$).

Before considering the complexity of the algorithm, we need the following combinatorial result of Robson [72].

Proposition 2.2.1 *Let $R_d(m, n)$ be the number of connected induced subgraphs of order m which are contained in a graph of order n and degree upper bounded by d. Then:*

$$R_d(m, n) = \mathrm{O}\left(\frac{n}{m}\left(\frac{(d-1)^{d-1}}{(d-2)^{d-2}}\right)^m\right).$$

In particular, $R_6(2h, 2k) = \mathrm{O}(\frac{k}{h}(\frac{5^5}{4^4})^{2h})$.

Theorem 2.2.1 *Algorithm $\mathcal{A}_C$ solves vertex cover in $\mathrm{O}(1.2759^k k^{1.5} + kn)$ time, where n is the number of nodes in the graph and k is the size of the vertex cover.*

Proof. The correctness of the algorithm follows from the correctness of algorithm $\mathcal{A}$ and from equation (2.1).

Consider the complexity of $\mathcal{A}_C$. The cost of the preliminary phase is not modified. One can both filter out small connected components and search in the database in $O(k)$ time (with a careful implementation of the database). Thus the cost associated to each branching is the same as in $\mathcal{A}$, that is $O(k^{1.5})$. The complexity of $\mathcal{A}_C$ can be derived from the analysis of Section 2.1 by replacing $\binom{2k}{2h}$ with $R_6(2h, 2k)$. It follows that:

$$\alpha = \frac{\log(c)}{\log(c) + 2\log(\frac{5^5}{4^4})} > 0.04867.$$

Thus the complexity of $\mathcal{A}_C$ is $O(c^{(1-\alpha)k}k^{1.5} + kn) = O(1.2759^k k^{1.5} + kn)$. □

2.3 A Further Refinement

In Section 2.2 we showed that restricting the class of graphs which are stored in the database, without increasing the branching factor, leads to a faster algorithm. In particular, the algorithm of Section 2.2 stores connected induced subgraphs only (instead of general induced subgraphs). Now we show how it is possible to store only connected induced subgraphs of degree lower bounded by two. This leads to a $O(1.2745^k k^4 + kn)$ algorithm $\mathcal{A}'_C$ for vertex cover.

Nodes of degree zero can be safely removed (since they do not belong to any minimum vertex cover). Then we just need to take care of nodes of degree one. For Lemma 1.2.1, these nodes can be filtered out in linear time. Observe that, if one starts with a problem (F, h) which satisfies the order-condition, the graph obtained after removing nodes of degree at most one satisfies the order-condition too. Moreover this filtering out does not modify the branching factor of the algorithm.

Before considering the complexity of the algorithm, we need the following combinatorial result of Robson [73].

Proposition 2.3.1 *Let $R_{2,d}(m,n)$ be the number of connected induced subgraphs of order m and degree lower bounded by two which are contained in a graph of order n and degree upper bounded by d. Let moreover:*

$$r_d = \max_{x \in X} \left\{ 2^{-x_0} \prod_{i=0}^{d-1} \left(\frac{\binom{d-1}{i}}{x_i} \right)^{x_i} \right\},$$

where

$$X = \left\{ x = (x_0, x_1 \dots x_{d-1}) \in \mathbb{R}^d : x_i \geq 0, \sum_i x_i = \sum_i i x_i = 1 \right\}.$$

Then:

$$R_{2,d}(m,n) = \mathrm{O}(r_d^m \, n \, poly_d(m)),$$

where, for any fixed d, $poly_d(m)$ is a polynomial of m.

Robson did not specified the value of $poly_d(m)$ since it was not relevant for his purpose. Following his proof, it is not hard to derive that $poly_d(m)$ is $\mathrm{O}(m^d)$. With a more careful analysis, one obtains that $poly_d(m)$ is $\mathrm{O}(m^{\frac{d-3}{2}})$. In particular, $R_{2,6}(2h,2k)$ is $\mathrm{O}(r_6^{2h} poly_6(h)k) = \mathrm{O}(9.927405^{2h} h^{1.5} k)$.

Theorem 2.3.1 *Algorithm $\mathcal{A}'_C$ solves vertex cover in $\mathrm{O}(1.2745^k k^4 + kn)$ time, where n is the number of nodes in the graph and k is the size of the vertex cover.*

Proof. The correctness of the algorithm follows from the correctness of algorithm $\mathcal{A}_C$ and from simple considerations.

The complexity analysis of the algorithm is analogous to the complexity analysis of algorithm $\mathcal{A}_C$. The main difference is that $R_6(2h,2k)$ is replaced by $R_{2,6}(2h,2k)$. It follows that:

$$\alpha = \frac{\log(c)}{\log(c) + 2\log(r_6)} > 0.05282.$$

Thus the time complexity of the algorithm is $\mathrm{O}(c^{(1-\alpha)k} k^{2.5} k^{1.5} + kn) = \mathrm{O}(1.2745^k k^4 + kn)$.

□

Chapter 3

Clique

In this chapter we provide improved algorithms for the detection of cliques of fixed order k, both in the sparse as well as in the dense case. We moreover present faster algorithms for the detection of *diamonds* and of *directed simple cycles* of length four.

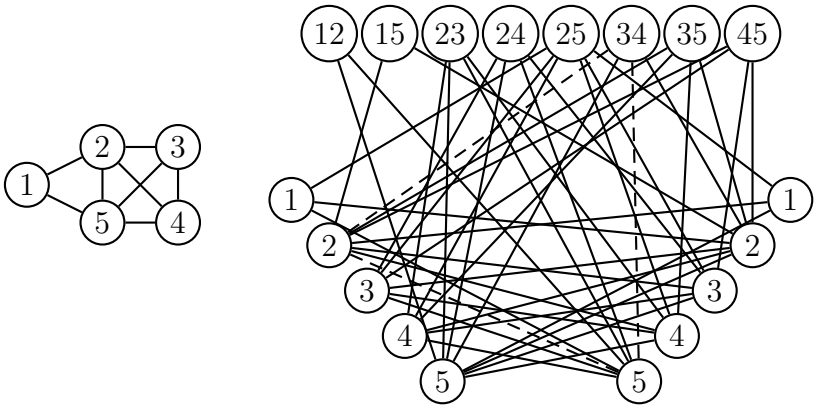
3.1 Dense Case

In this section we present our $O(n^{\omega(\lfloor k/3 \rfloor, \lceil (k-1)/3 \rceil), \lceil k/3 \rceil)})$ algorithm for clique problem in dense graphs.

First we recall how the algorithm of Nešetřil and Poljak [61] for the same problem works. If $k = 3h$ for some $h \in \mathbb{N}$, one creates an auxiliary graph $\widetilde{G}$ which has a node for each h-clique of G and an edge between a pair of nodes if and only if the corresponding nodes in G form a $(2h)$-clique. The graph G contains a k-clique if and only if $\widetilde{G}$ contains a triangle. Since $\widetilde{G}$ has $O(n^h)$ nodes, a triangle in it can be detected in $O(n^{\omega h})$ time by using the algorithm of Itai and Rodeh [44]. If k is not divisible by three, one applies the following fact. A node v is contained in a k-clique if and only if the graph $G[N(v)]$ induced on G by the neighborhood $N(v)$ of v, contains a $(k-1)$-clique. This implies that one can detect a k-clique of G by applying an algorithm to detect a $(k-1)$-clique in each graph $G[N(v)]$, $v \in V$. Thus one can detect a k-clique in $O(n^{\alpha(k)})$ time, where $\alpha(k) = \lfloor k/3 \rfloor \omega + k \pmod 3$.

The idea behind our algorithm for clique problem is to allow for different orders of the

Figure 17 A graph G (on the left) with the corresponding auxiliary graph $\widetilde{G}$. The nodes of $\widetilde{G}$ are labelled with the corresponding subset of nodes of G. One of the $\binom{4}{1,1,2} = 12$ triangles corresponding to the clique $\{2,3,4,5\}$ is pointed out via dashed lines.



sub-cliques, so that the case $k \pmod 3 \neq 0$ is not treated separately. Let k_1, k_2 and k_3 be equal to $\lfloor k/3 \rfloor$, $\lceil (k-1)/3 \rceil$ and $\lceil k/3 \rceil$ respectively (notice that $k = k_1 + k_2 + k_3$). The algorithm works as follows. It first creates the following 3-partite auxiliary graph $\widetilde{G}$. The nodes of $\widetilde{G}$ are partitioned into sets V_i, for each $i \in \{1,2,3\}$, where the nodes in V_i are the cliques of order k_i of G. A node $u \in V_i$ is connected with a node $v \in V_j$, if $i \neq j$ and the nodes of u and v induce a $(k_i + k_j)$-clique in G. The answer is *yes* if and only if $\widetilde{G}$ contains a triangle. Figure 17 shows a graph G with the corresponding auxiliary graph $\widetilde{G}$. We need now to show how the algorithm searches for a triangle in $\widetilde{G}$. In fact, since $\widetilde{G}$ is 3-partite, a procedure exists which is more efficient than the algorithm of Itai and Rodeh. For every pair of nodes $u \in V_1$ and $v \in V_3$, the algorithm computes the number $P(u,v)$ of 2-length paths from u to v through a node in V_2. Clearly, graph $\widetilde{G}$ contains a triangle if and only if there is a pair of adjacent nodes u and v, $u \in V_1$ and $v \in V_3$, such that $P(u,v) > 0$.

Theorem 3.1.1 *The algorithm above determines whether an undirected graph G of n nodes contains a clique of k nodes in $\mathrm{O}(n^{\beta(k)}) = \mathrm{O}(n^{\omega(\lfloor k/3 \rfloor, \lceil (k-1)/3 \rceil, \lceil k/3 \rceil)})$ time, for every fixed $k \geq 3$.*

Proof. Assume that G contains a k-clique $\{v_1, v_2 \ldots v_k\}$. Thus the partitions V_1, V_2 and

V_3 of $\widetilde{G}$ contain the nodes $w_1 = \{v_1, v_2 \dots v_{k_1}\}$, $w_2 = \{v_{k_1+1}, v_{k_1+2} \dots v_{k_1+k_2}\}$ and $w_3 = \{v_{k_1+k_2+1}, v_{k_1+k_2+2} \dots v_k\}$ respectively. Moreover w_1, w_2 and w_3 are pairwise adjacent. Thus $\widetilde{G}$ contains a triangle. Assume now that $\widetilde{G}$ contains a triangle $\{w_1, w_2, w_3\}$. Let $T = \bigcup_i w_i$. Since the graph is 3-partite, the nodes w_i must belong to distinct partitions. Moreover they cannot share a common node v of G. Thus $|T| = k_1 + k_2 + k_3 = k$. Every two distinct nodes of T are adjacent in G. Thus T is a subset of k pairwise adjacent nodes of G, that is a k-clique of G.

Consider now the time complexity of the algorithm. Partition V_i contains $\mathrm{O}(n^{k_i})$ nodes, $i \in \{1, 2, 3\}$. The cost of the algorithm is upper bounded by the cost of computing the number of two length paths from the nodes of V_1 to the nodes of V_3 through the nodes of V_2, that is the cost of multiplying the $n^{k_1} \times n^{k_2}$ adjacency matrix of the nodes in V_1 with the nodes in V_2 by the $n^{k_3} \times n^{k_2}$ adjacency matrix of the nodes in V_2 with the nodes in V_3. Thus the time complexity of the algorithm is $= \mathrm{O}(n^{\omega(k_1,k_2,k_3)}) = \mathrm{O}(n^{\beta(k)})$. □

If the rectangular matrix multiplication required by the algorithm is carried out via the straightforward partition into square blocks and fast square matrix multiplication, one obtains the same time complexity of Nešetřil and Poljak [61]:

$$\beta(k) \leq (k_3 - k_1) + (k_2 - k_1) + \omega(k_1, k_1, k_1) = k \pmod 3 + k_1\omega = \alpha(k).$$

An asymptotically better bound can be obtained, when $k \pmod 3 \neq 0$, by using more sophisticated fast rectangular matrix multiplication algorithms [16, 43].

Consider first the case $k = 3h + 1$, where $h \in \mathbb{N}$. In this case $\omega(k_1, k_2, k_3) = (h, h, h + 1)$ and we have to compare the bounds in [43] for $\omega(1, 1, 1 + 1/h)$ with the trivial bound $\omega(1, 1, 1 + 1/h) \leq 1/h + \omega$, which follows from partitioning the matrix into square submatrices and square matrix multiplication.

If $r \geq 1.171$, the bound for $\omega(1, 1, r)$ given in [43] (which is not expressed via a closed formula) is superior to the trivial bound $\omega(1, 1, r) \leq r - 1 + \omega$. This implies that $\beta(3h + 1)$

is strictly less than $\alpha(3h+1)$ for any positive $h \leq 5$:

$$\beta(3h+1) = \omega(h,h,h+1) = h\omega(1,1,1+1/h) < h(1/h+\omega) = \alpha(3h+1).$$

Consider now the case $k = 3h+2$, where $h \in \mathbb{N}$. In this case $\omega(k_1,k_2,k_3) = (h,h+1,h+1)$ and we have to compare the trivial bound for $\omega(1,1,h/h+1)$ with the following bound [16, 43]. For $0 \leq r \leq 1$:

$$\omega(1,1,r) \leq \begin{cases} 2+o(1) & \text{if } 0 \leq r \leq \delta = 0.294, \\ \omega + \frac{(1-r)(2-\omega)}{1-\delta} & \text{if } \delta < r \leq 1. \end{cases}$$

In our case, r is at least $1/2 > \delta$, so the second bound applies. We have to compare $\alpha(3h+2) = 2+\omega h$ with $\beta(3h+2) = (h+1)\omega(1,1,h/(h+1))$. Thus

$$\begin{aligned} (h+1)\omega(1,1,\frac{h}{h+1}) &\leq (h+1)(\omega + \frac{2-\omega}{(h+1)(1-\delta)}) \\ &= 2+\omega h - \frac{(\omega-2)\delta}{1-\delta} \\ &= \alpha(3h+2) - \frac{(\omega-2)\delta}{1-\delta}. \end{aligned}$$

This implies that $\beta(3h+2)$ is strictly less than $\alpha(3h+2)$ for every positive h. In particular, the improvement obtained is lower bounded by $\frac{(\omega-2)\delta}{1-\delta}$.

Summing up, our algorithm is asymptotically faster than the algorithm of Nešetřil and Poljak if $k \pmod 3 = 1$ and $k \leq 16$, or $k \pmod 3 = 2$. In Table 4 the complexity of our algorithm is compared with the complexity of the algorithm of Nešetřil and Poljak for $4 \leq k \leq 7$.

k	Previous best [61]	This section
4	$O(n^{3.376})$	$O(n^{3.334})$
5	$O(n^{4.376})$	$O(n^{4.220})$
6	$O(n^{4.751})$	$O(n^{4.751})$
7	$O(n^{5.751})$	$O(n^{5.714})$

Table 4: Running time comparison for clique problem in dense graphs.

3.1.1 The (Induced) Subgraph Problem

The *(induced) subgraph problem* consists in determining whether a graph G of n nodes contains an (induced) subgraph F of k nodes. Nešetřil and Poljak [61] showed how to build an auxiliary graph $\widetilde{G}$ of kn nodes such that $\widetilde{G}$ contains a k-clique if and only if G contains an (induced) F (the reduction is described in Section 1.3). If we combine their reduction with our algorithm for clique detection, we obtain a $O(n^{\beta(k)})$ algorithm for the (induced) subgraph problem.

Corollary 3.1.1 *The algorithm above determines whether an undirected graph G with n nodes contains an (induced) subgraph F of k nodes in $O(n^{\alpha(k)})$ time, for every fixed $k \geq 3$.*

3.1.2 Counting Cliques

Consider the problem of counting the number $K_k(v)$ of k-cliques in which each node v of G is contained. Note that many triangles in $\widetilde{G}$ may correspond to the same k-clique of G. More precisely, the number of distinct triangles of $\widetilde{G}$ which correspond to the same k-clique of G is equal to the number of ways in which we can partition a set of cardinality k in three subsets of cardinality k_1, k_2 and k_3 respectively, which is $\binom{k}{k_1,k_2,k_3} = \frac{k!}{k_1!k_2!k_3!}$. The algorithm of Section 3.1 can be easily adapted to count, in $O(n^{\beta(k)})$ time, the number $T(w)$ of triangles in which each node w of $\widetilde{G}$ is contained. Let $W_i(v)$, for each $i \in \{1, 2, 3\}$, be the set of nodes of W_i corresponding to the k_i-cliques which contain node v. It is not hard to show that the sum of $T(w)$ over $W_1(v)$ is equal to $K_k(v)$, multiplied by the number of ways in which we can partition a set of cardinality (k-1) in three subsets of cardinality (k_1-1), k_2 and k_3 respectively:

$$\sum_{w \in W_1(v)} T(w) = \binom{k-1}{k_1 - 1, k_2, k_3} K_k(v). \tag{3.1}$$

Then we can compute $K_k(v)$, for every $v \in V$, in $O(n^{\beta(k)})$ steps.

Proposition 3.1.1 *The algorithm above counts the number of k-cliques in which each node of an undirected graph G of n nodes is contained, in $O(n^{\beta(k)})$ time.*

3.2 Sparse Case

In this section we are concerned with the detection of cliques in a *sparse* undirected graph G. In particular, we want to develop efficient algorithms which depend on the number e of edges only.

By simply combining the algorithm of Section 3.1 with the algorithm of Kloks et al. [49], one already obtains a $O(e^{\frac{\beta(k)\beta(k-1)}{\beta(k)+\beta(k-1)-1}})$ algorithm for clique detection in sparse graphs, which improves on the previous best algorithm [49], whose complexity is $O(e^{\frac{\alpha(k)\alpha(k-1)}{\alpha(k)+\alpha(k-1)-1}})$. However, the $O(n^{\beta(k)})$ running time obtained by the dense case algorithm is superior for some values of k if the graph is sufficiently dense. This is because for some values of k, $\beta(k) < \beta(k-1)+1$ and thus $\frac{\beta(k)\beta(k-1)}{\beta(k)+\beta(k-1)-1} > \frac{\beta(k)}{2}$. The natural question arises, whether there exists an algorithm of running time $O(e^{\beta(k)/2})$, for any fixed $k \geq 3$. We now give a positive answer to this for the case $k \geq 6$.

Erdős [32] proved the following lemma (see also [9]).

Lemma 3.2.1 *Let $e = \binom{s}{2} + t$ be the number of edges of a graph G, where $s, t \in \mathbb{N}$ and $s < t$. Then G contains at most $\binom{s}{k} + \binom{t}{k-1}$ cliques of order $k \geq 3$ and this upper bound is tight.*

Thus there are at most $O(e^{k/2})$ k-cliques in G. It turns out that all such cliques can be enumerated within the same time bound. Consider the following algorithm. Let L denote the set of nodes with degree smaller than $\Delta = \sqrt{e}$, and let H denote the set of the remaining nodes. The algorithm first enumerates all the cliques which contain at least one node in L. To do that, it simply enumerates all the $(k-1)$-cliques of $G[N(v)]$, for every $v \in L$ (via the trivial algorithm which considers all the subsets of $(k-1)$ nodes). Then it enumerates, via the trivial algorithm, all the k-cliques formed by nodes in H only.

Proposition 3.2.1 *The algorithm above enumerates all the k-cliques of a graph G with e edges in $O(e^{k/2})$ time, for every fixed $k \geq 2$.*

Proof. The correctness of the algorithm is trivial.

The cliques containing at least one node in L can be enumerated in $O(\sum_{v \in L} deg(v)^{k-1}) = O(e\,\Delta^{k-2})$ time. The cardinality of H is bounded by $|H| \leq 2\,e/\Delta$. This implies that the cliques formed by nodes in H only can be enumerated in $O((e/\Delta)^k)$ time. Since $\Delta = \sqrt{e}$, the complexity of the algorithm is $O(e^{k/2})$. □

Notice that we can label each edge of G with the number of k-cliques to which it belongs to within the same time bound. Assume that the set of nodes is totally ordered. As one enumerates the k-cliques, one can generate a list U of ordered k-tuples $T = (t_1, \ldots, t_k)$, which represent the nodes of each clique. For each T one has to augment the label of edge $\{t_i, t_j\}$, for each $1 \leq i < j \leq k$, by one. To do this in linear time, we consider all the possible choices of i and j, and we generate lists $U_{i,j}$ which consist of the pairs (t_i, t_j) for each $T \in U$. Next we lexicographically sort each $U_{i,j}$ with radix sort in linear time. Then we scan each list and add one to the edge label corresponding to each scanned pair (t_i, t_j). Notice that this can be done in linear time in the number of edges and in the size of the lists. We thus have the following corollary.

Corollary 3.2.1 *The algorithm above labels each edge $\{v, w\}$ of an undirected graph G of e edges with the number of k-cliques, $k \geq 2$, which $\{v, w\}$ belongs to, in time $O(e^{k/2})$.*

A $O(e^{\beta(k)/2})$ algorithm for clique problem, $k \geq 6$, straightly derives from the previous result and from the algorithm described in Section 3.1. As in the dense case, the algorithm builds the 3-partite auxiliary graph $\widetilde{G}$ and it looks for a triangle in it. Remember that the partition V_i, $i \in \{1, 2, 3\}$, is formed by the k_i-cliques of G, with k_1, k_2 and k_3 being $\lfloor k/3 \rfloor$, $\lceil (k-1)/3 \rceil$ and $\lceil k/3 \rceil$ respectively. These cliques are enumerated via the algorithm of Proposition 3.2.1.

Theorem 3.2.1 *The algorithm above determines whether an undirected graph G of e edges contains a clique of order k, $k \geq 6$, in $O(e^{\beta(k)/2})$ time.*

Proof. The correctness is a straightforward consequence of the correctness of Proposition 3.2.1 and of Theorem 3.1.1.

From Proposition 3.2.1, the set V_i, $i \in \{1,2,3\}$, has cardinality $O(e^{k_i/2})$, and can be created within the same time bound. The cost to detect a triangle in $\widetilde{G}$ is bounded by the time required to multiply the $e^{k_1/2} \times e^{k_2/2}$ adjacency matrix of the nodes in V_1 with the nodes in V_2 by the $e^{k_2/2} \times e^{k_3/2}$ adjacency matrix of the nodes in V_2 with the nodes in V_3. This multiplication costs $O(e^{\omega(k_1/2,k_2/2,k_3/2)}) = O(e^{\beta(k)/2})$. □

Recall that the algorithm of Kloks et al. (in the improved version presented in this section) runs in time $O(e^{\beta(k)\beta(k-1)/(\beta(k)+\beta(k-1)-1)})$. It is easy to see that $\frac{\beta(k)}{2} \leq \frac{\beta(k)\beta(k-1)}{\beta(k)+\beta(k-1)-1}$ for every $k \geq 6$. Furthermore, for every $k \geq 6$, the complexity of our algorithm is never inferior to the $O(n^{\beta(k)})$ complexity of the dense case algorithm. By now, this is not the case for $3 \leq k \leq 5$. Here, the fastest detection algorithm is the $O(e^{\beta(k)\beta(k-1)/(\beta(k)+\beta(k-1)-1)})$ algorithm. Whether there exists a $O(e^{\beta(k)/2})$ algorithm for $3 \leq k \leq 5$ is an interesting open problem. In Table 5 the complexity of our algorithm is compared with the complexity of the algorithm of Kloks et al. [49] for $4 \leq k \leq 7$.

k	Previous best [49]	This section
4	$O(e^{1.688})$	$O(e^{1.682})$
5	$O(e^{2.188})$	$O(e^{2.147})$
6	$O(e^{2.559})$	$O(e^{2.376})$
7	$O(e^{2.876})$	$O(e^{2.857})$

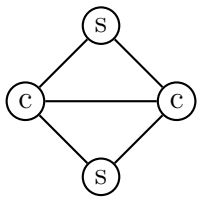
Table 5: Running time comparison for clique problem in sparse graphs.

3.3 Diamonds

In this section we consider the detection of an induced diamond. A *diamond*, which is depicted in Figure 18, is a graph with four nodes, which results from a 4-clique via the deletion of one edge. Detecting diamonds is important since maximum clique problem is solvable in polynomial time in diamond-free graphs.

A diamond can be detected with the algorithm of Section 3.1.1 in $O(n^{\beta(4)}) = O(n^{3.334})$

Figure 18 A diamond. Side and central nodes are labelled with s and c respectively.



time. Interestingly, there is a faster algorithm for this problem. Kloks et al. [49] showed that a diamond can be detected in $O(n^\omega + e^{3/2})$ steps (see Section 1.4). Here we present an algorithm for diamond detection which runs in $O(e^{3/2})$ steps. Our algorithm does not make use of fast matrix multiplication. We assume that the graph is represented via adjacency lists.

We again use the technique to decompose the set of nodes into high and low degree nodes H and L. The value of Δ will be determined in the sequel. A diamond contains two nodes of degree three and two of degree two. Let us call the nodes of the first kind *central nodes*, and the other two nodes *side nodes*. First we look for a diamond which contains a low degree central node. For every low degree node v, we create the adjacency matrix of $G[N(v)]$, we compute the connected components in $G[N(v)]$ and we check if they are all cliques. If not, v is contained in a diamond for Proposition 1.4.1 and we can detect it in $O(deg(v)^2)$ time. The complexity of this step is bounded by the time required to create the adjacency matrices. We can do that in $O(e + \sum_{v \in L} deg(v)^2) = O(e\Delta)$ steps in the following way. We create an n-elements vector R that we initialize to zero. Then the algorithm proceeds through n-rounds. In the v-th round we fill in all the rows of the adjacency matrices which correspond to the node v. First we set to one all the entries of R corresponding to the neighbors of v. Now the vector R is equal to the v-th row of the adjacency matrix of G (which is not available). Then for each neighbor u of v, we detect the row corresponding to v in the adjacency matrix of $G[N(u)]$, and we fill in that row in linear time by using the vector R. At the end of each round we reset the non-zero entries of R. This procedure has

a linear cost in the number of edges and in the size of the adjacency matrices created.

If no diamond is detected in the first step, we look for a diamond which contains a low degree side node. It follows from Corollary 3.2.1 that we can label each edge with the number of triangles of G in which it is contained in $O(e^{1.5})$ steps. Consider a low degree node v. The graph $G[N(v)]$ is a disjoint union of cliques. The node v belongs to a diamond if and only if there exists an edge in a h-clique of $G[N(v)]$ which belongs to at least h triangles. This second step costs $O(e^{1.5} + \sum_{v \in L} deg(v)^2) = O(e^{1.5} + e\Delta)$ time.

The diamonds not yet considered are formed by high degree nodes only. The size of H is bounded by $|H| \leq 2\,e/\Delta$. Using the same approach of the first step, we can detect a diamond of this kind in $O(e + \sum_{v \in H} deg_H(v)^2) = O(e + e^2/\Delta)$ steps, where $deg_H(v)$ is the number of high degree neighbors of v.

The above described procedure runs in time $O(e^{1.5} + e\Delta + e^2/\Delta)$. This complexity is minimized by setting $\Delta = \sqrt{e}$.

Theorem 3.3.1 *The algorithm above determines whether an undirected graph G of e edges contains an induced diamond, in time* $O(e^{1.5})$.

3.3.1 Counting Diamonds

Consider now the problem to count the number d of induced subgraphs of G isomorphic to a diamond. Kloks et al. [49] described an algorithm to count the number k_4 of 4-cliques in G. Considering the results of Section 3.1, the running time of their algorithm is $O(e^{\beta(4)\beta(3)/(\beta(4)+\beta(3)-1)})$. They moreover noticed that:

$$t = \sum_{\{u,w\} \in E} \binom{A^2[u,w]}{2} = 6k_4 + d,$$

where A is the 0-1 adjacency matrix of G ($A^2[u,w]$ is equal to the number of 2-length paths between u and w). Thus the value of d can be determined in

$$O(n^\omega + e^{\beta(4)\beta(3)/(\beta(4)+\beta(3)-1)})$$

steps, where $O(n^\omega)$ is the time required to compute A^2.

A better bound can be obtained by using Corollary 3.2.1. We can label in $O(e^{1.5})$ time each edge $\{u, w\} \in E$ with the number $T(u, w)$ of triangles in which that edge is contained. As $T(u, v)$ is equal to $A^2[u, w]$ for any $\{u, w\} \in E$, we can compute t within the same time bound. Then the value of d can be computed in $O(e^{\beta(4)\beta(3)/(\beta(4)+\beta(3)-1)}) = O(e^{1.682})$ steps.

Proposition 3.3.1 *The algorithm above counts the number of induced diamonds contained in an undirected graph G of e edges in $O(e^{\beta(4)\beta(3)/(\beta(4)+\beta(3)-1)})$ time.*

3.4 Simple Directed 4-Cycles

Let $G = (N, A)$ be an arbitrary directed graph G with n nodes and e edges. In this section we present an algorithm to detect a C_4, a *directed simple cycle* of length 4, in G in $O(n^{1/\omega}e^{2-2/\omega})$ time. This improves upon the currently fastest methods for $\alpha \in \left(\frac{2}{4-\omega}, \frac{\omega+1}{2}\right)$, where $e = n^\alpha$.

Again, the set of nodes is decomposed in two subsets: a set L of nodes with degree smaller than Δ (low degree nodes), and the set H of the remaining nodes (high degree nodes). The value of Δ will be determined in the sequel.

We first describe how to detect a C_4 which contains two opposite low degree nodes, i.e., it is of the form v_1, v_2, v_3, v_4, where v_1 and v_3 are members of L. Following [2], we first compute all paths of length 2 which have an intermediate node in L. These paths can be constructed in $O(e\Delta)$ time. Next one sorts these paths with respect to their start and end point in lexicographic and reverse lexicographic order with radix sort in linear time. With this at hand, it is then straightforward to detect whether there exist two such paths, which form a simple directed cycle of length four. This procedure runs in $O(e\Delta)$ steps.

Since $\sum_{v \in N} \deg(v) = 2\,e$, one has that the number of high degree nodes $|H|$ is bounded by $|H| \leq 2\,e/\Delta$. Thus a C_4 which contains only high degree nodes can be detected in $O((e/\Delta)^\omega)$ steps via fast matrix multiplication.

The other C_4 are of the form

1. C_{LHHH}: three high degree nodes followed by one low degree node;

2. C_{LLHH}: a pair of consecutive low degree nodes followed by a pair of high degree nodes.

Let M_{ABC} be an integer matrix such that, given a node $v \in A$ and a node $w \in C$, $M_{ABC}[v, w]$ is the number of 2-length directed simple paths from v to w which pass through a node in B, where $A, B, C \in \{L, H\}$. Since there are at most $2\,e/\Delta$ high degree nodes, one can compute M_{HHH} in time $\mathrm{O}((e/\Delta)^{\omega})$.

One can compute the matrix M_{HHH} in time $\mathrm{O}((e/\Delta)^{\omega})$. The matrix M_{HHL} can be computed in time $\mathrm{O}((e/\Delta)^{\omega(1,1,\log n/\log(e/\Delta))})$.

With these matrices M_{HHH} and M_{HHL} at hand, we can now check whether there exists a C_4 of type 1 or 2. To do so, one generates again all 2-paths with intermediate node in L. For each such path, one then checks whether its start-node u and its end-node v have a positive entry $M_{HHH}[u, v]$ or $M_{HHL}[u, v]$.

The above described procedure runs in time $\mathrm{O}(e\,\Delta + (e/\Delta)^{\omega(1,1,\log n/\log(e/\Delta))})$. This complexity is minimized when $e\,\Delta = (e/\Delta)^{\omega(1,1,\log n/\log(e/\Delta))}$. The exponent $\omega(1, 1, r)$ is bounded by $\omega(1, 1, r) \leq r - 1 + \omega$. Via this upper bound one obtains $\Delta = n^{1/\omega} e^{1-2/\omega}$ and thus a running time of $\mathrm{O}(n^{1/\omega} e^{2-2/\omega})$.

Theorem 3.4.1 *The algorithm above detects a C_4 in a directed graph with n nodes and e edges in time $\mathrm{O}(n^{1/\omega} e^{2-2/\omega})$.*

This running time is an asymptotic improvement of the $\mathrm{O}(n^{\omega})$ and $\mathrm{O}(e^{1.5})$ bounds if $e = n^{\alpha}$ for $\alpha \in (\frac{2}{4-\omega}, \frac{\omega+1}{2})$. The currently best algorithm for matrix multiplication shows that ω is bounded by $\omega < 2.375477$ [17]. This means that our algorithm is the currently fastest algorithm for $\alpha \in [1.2312, 1.6877]$.

3.4.1 Remarks

Limitations to detecting larger cycles To speed up the detection of a C_k for $k \geq 5$ one cannot use the same approach. Already to detect a C_5 with the same idea, one would

have to spend at least $O((e/\Delta)^\omega)$ steps to detect a C_5 among the high degree nodes and in addition one would have to generate all 3-paths in the low degree nodes, of which there might be $e\Delta^2$ many. Thus the running time would be $\Omega(e^{1+2/3})$, which is not superior to the $O(e^{1+2/3})$ algorithm of Alon et al. [2].

Using fast rectangular matrix multiplication Above we used the simple upper bound $\omega(1,1,r) \leq r - 1 + \omega$ to readily compute the optimal value of Δ in terms of n and e. In this way we could also state a closed formula for the worst case complexity of our algorithm. Huang and Pan [43] described a fast method for rectangular matrix multiplication. The bound on $\omega(1,1,r)$ which results from their algorithm is superior to the $r - 1 + \omega$ bound that we applied. However, it is not expressed via a closed formula. We numerically found that using Huang and Pan's fast rectangular matrix multiplication algorithm, our method is asymptotically faster than $O(n^\omega)$ and $O(e^{1.5})$ for any α in the interval $[1.2117, 1.6877]$.

Chapter 4

Decremental Clique and Applications

The clique problem consists in determining whether an undirected graph G of order n contains a clique of order k. In this chapter we are concerned with the *decremental clique* problem, where the property of containing a k-clique is dynamically checked during deletions of nodes. In Section 4.1 we present a data structure which allows to decrementally maintain the number $K_k(v)$ of k-cliques in which each node v of an undirected graph G is contained, during deletions of nodes. The preprocessing time is $\mathrm{O}(n^{\beta(k)})$, that is the time required to multiply a $n^{\lfloor k/3 \rfloor} \times n^{\lceil (k-1)/3 \rceil}$ matrix by a $n^{\lceil (k-1)/3 \rceil} \times n^{\lceil k/3 \rceil}$ matrix. The query time is $\mathrm{O}(1)$ and the amortized update time per deletion is $\mathrm{O}(n^{\widetilde{\beta}(k)}) = \mathrm{O}(n^{\beta(k)-0.8})$. This data structure can be easily adapted to solve the decremental clique problem. In fact, the answer to the problem is *yes* if and only if there is a node v in the current graph such that $K_k(v) > 0$.

Our data structure naturally applies to *filtering* for the binary constraints satisfaction problem. In particular, we show how to speed up the filtering based on two important local consistency properties: the *inverse consistency* and the *max-restricted path consistency*. In Section 4.2 we describe a $O(n^\ell d^{\widetilde{\beta}(\ell)+1})$ ℓ-inverse consistency based filtering algorithm, which makes use of the decremental algorithm of Section 4.1. This improves on the $O(n^\ell d^\ell)$ algorithm of Debruyne [20] for every $\ell \geq 3$.

In Section 4.3.1 we describe a max-restricted path consistency based filtering algorithm, of

$O(end^3)$ time complexity and $O(ed)$ space complexity. In Section 4.3.2 we present another max-restricted path consistency based filtering algorithm of time complexity $O(end^{2.575})$ and space complexity $O(end^2)$. This algorithm makes use of a variant of the decremental algorithm of Section 4.1. The previously best max-restricted path consistency based filtering algorithm is the algorithm of Debruyne and Bessiere [21], of time complexity $O(end^3)$ and space complexity $O(end)$. The experiments performed on *random consistency graphs* suggest that the algorithm of Section 4.3.1 is faster than the algorithm of Debruyne and Bessiere for problems of low *density* and high or low *tightness*.

4.1 Decremental Clique

In this section we present a data structure which allows to decrementally maintain the number $K_k(v)$ of k-cliques in which each node v of an undirected graph G is contained, during deletions of nodes. The data structure implements a dynamic version of the static algorithm of Section 3.1.2.

We use the same notation as in Section 3.1.2. In particular, $\widetilde{G} = (\{W_1, W_2, W_3\}, E)$ is the 3-partite auxiliary graph and $T(w)$ is the number of triangles in which node w of $\widetilde{G}$ is contained. Moreover, $W_i(v)$ is the subset of nodes of W_i which contain node v of G. The current values of $T(\cdot)$ and $K_k(\cdot)$ can be initially computed in $O(n^{\beta(k)})$ time via the algorithm of Section 3.1.2. We assume that $K_k(\cdot)$ is updated after each deletion. This way, the query time is $O(1)$. We have now to show how to dynamically perform updates. Let u be the deleted node. The idea is to update the value of $T(w)$, for each w in W_1, and then update $K_k(v)$, for each v in w, following equation (3.1). In more details, the deletion of u corresponds to the deletion of the subsets of nodes $W_1(u)$, $W_2(u)$ and $W_3(u)$ in W_1, W_2 and W_3 respectively. Notice that $W_i(u)$, $i \in \{1, 2, 3\}$, contains $O(n^{k_i-1})$ nodes. Since two nodes of $\widetilde{G}$ which contain the same node u of G cannot belong to the same triangle, one can safely consider the effects of the deletion of each node in $W_i(u)$, $i = 1, 2, 3$, separately. First of all,

for each $w \in W_1(u)$, one sets $T(w)$ to zero (in linear time). Then one considers the deletion of the nodes in $W_2(u)$ (the deletion of the nodes in $W_3(u)$ is treated analogously). For each $w_1 \in W_1$ and for each $w_2 \in W_2(u)$, one needs to decrease $T(w_1)$ of the number $T(w_1, w_2)$ of triangles in which both nodes w_1 and w_2 are contained (at the same time). The value of $T(w_1, w_2)$ is zero if w_1 and w_2 are not adjacent, and it is equal to the number $P_{1,3,2}(w_1, w_2)$ of 2-length paths from w_1 to w_2 through a node in W_3 otherwise. Then, to update $T(\cdot, \cdot)$, one needs to compute $P_{1,3,2}(w_1, w_2)$, for each $w_1 \in W_1$ and for each $w_2 \in W_2(u)$. A simple-minded approach is to compute the number of such paths from scratch, by multiplying the $n^{k_1} \times n^{k_3}$ adjacency matrix of the nodes in W_1 with the nodes in W_3 by the $n^{k_3} \times n^{k_2-1}$ adjacency matrix of the nodes in W_3 with the nodes in $W_2(u)$. This costs $\mathrm{O}(n^{\omega(k_1,k_3,k_2-1)})$ per update. With this approach, the query time is $\mathrm{O}(1)$.

It is possible to reduce the update time, while maintaining a constant query time, in the following way. The idea is to maintain a lazy value $\widetilde{P}_{1,3,2}(w_1, w_2)$ of $P_{1,3,2}(w_1, w_2)$, for each $w_1 \in W_1$ and $w_2 \in W_2$. Whenever a node w_3 is removed from W_3, instead of updating $\widetilde{P}_{1,3,2}$, one stores w_3 in a set D_3. When the cardinality of D_3 reaches a given threshold, one updates $\widetilde{P}_{1,3,2}$ and empties D_3. Clearly the current value of $P_{1,3,2}(w_1, w_2)$ depends on both $\widetilde{P}_{1,3,2}(w_1, w_2)$ and D_3. In more details, the algorithm initially sets $D_3 = \emptyset$ and $\widetilde{P}_{1,3,2} = P_{1,3,2}$. Let $\mu_3 \in [0, 1]$ be a parameter to be fixed later. When a node w_3 is removed from W_3, the algorithm adds it to D_3 and, if $|D_3| > n^{k_3-1+\mu_3}$, it executes the following steps:

- It updates $\widetilde{P}_{1,3,2}$ by subtracting from $\widetilde{P}_{1,3,2}(w_1, w_2)$ the number $\Delta P_{1,3,2}(w_1, w_2)$ of 2-length paths from w_1 to w_2 through a node in D_3.

- It sets $D_3 = \emptyset$.

Computing $\Delta P_{1,3,2}(\cdot)$ costs $\mathrm{O}(n^{\omega(k_1,k_3-1+\mu_3,k_2)})$ time, that is the time required to multiply the $n^{k_1} \times n^{k_3-1+\mu_3}$ adjacency matrix of the nodes in W_1 with the nodes in D_3 by the $n^{k_3-1+\mu_3} \times n^{k_2}$ adjacency matrix of the nodes in D_3 with the nodes in W_2. This is also an upper bound

on the cost to update $\widetilde{P}_{1,3,2}(\cdot)$. The deletion of $\mathrm{O}(n^{k_3-1})$ nodes in W_3 corresponds to each deletion of a node in G. This means that we update $\widetilde{P}_{1,3,2}$ every $\Omega(n^{\mu_3})$ deletions in G, with a $\mathrm{O}(n^{\omega(k_1,k_3-1+\mu_3,k_2)-\mu_3})$ amortized cost per update.

The current value of $P_{1,3,2}(w_1, w_2)$, is given by:

$$P_{1,3,2}(w_1, w_2) = \widetilde{P}_{1,3,2}(w_1, w_2) - \Delta P_{1,3,2}(w_1, w_2). \tag{4.1}$$

This value is used to update $T(\cdot)$, and then $K_k(\cdot)$. The value of $P_{1,3,2}(\cdot)$ can be computed in $\mathrm{O}(n^{\omega(k_1,k_3-1+\mu_3,k_2-1)})$ time, that is the time complexity of multiplying the $n^{k_1} \times n^{k_3-1+\mu_3}$ adjacency matrix of the nodes in W_1 with the nodes in D_3 by the $n^{k_3-1+\mu_3} \times n^{k_2-1}$ adjacency matrix of the nodes in D_3 with the nodes in $W_2(u)$. This is also an upper bound on the cost to update $T(\cdot)$ and $K_k(\cdot)$. Thus updating $K_k(\cdot)$ after the deletion of the nodes in $W_2(u)$ costs $\mathrm{O}(n^{\omega(k_1,k_3-1+\mu_3,k_2)-\mu_3} + n^{\omega(k_1,k_3-1+\mu_3,k_2-1)})$. Analogously, the cost of updating $K_k(\cdot)$ after the deletion of the nodes in $W_3(u)$ is $\mathrm{O}(n^{\omega(k_1,k_2-1+\mu_2,k_3)-\mu_2} + n^{\omega(k_1,k_2-1+\mu_2,k_3-1)})$, where $\mu_2 \in [0, 1]$ is a parameter which plays the same role as μ_3. The total amortized update time is minimized by choosing μ_2 and μ_3 properly. In particular, the optimal cost is $\mathrm{O}(n^{\widetilde{\beta}(k)})$, where

$$\begin{aligned}\widetilde{\beta}(k) = \min_{\mu_2,\mu_3} \max\{&\omega(k_1, k_3 - 1 + \mu_3, k_2) - \mu_3, \omega(k_1, k_3 - 1 + \mu_3, k_2 - 1),\\ &\omega(k_1, k_2 - 1 + \mu_2, k_3) - \mu_2, \omega(k_1, k_2 - 1 + \mu_2, k_3 - 1)\}.\end{aligned}$$

Theorem 4.1.1 *The algorithm above decrementally maintains the number of k-cliques in which each node of a graph G of n nodes is contained, during deletions of nodes, with a $\mathrm{O}(n^{\beta(k)})$ preprocessing time, a $\mathrm{O}(1)$ query time, and a $\mathrm{O}(n^{\widetilde{\beta}(k)})$ amortized update time.*

4.1.1 Bounds on the Complexity

In this section we will show how to fix μ_2 and μ_3 such as to minimize the update time. For this purpose, we will use the following bounds on $\omega(r, s, t)$ [16, 43]. For every $r < 1$:

$$\omega(1, 1, r) \leq \begin{cases} 2 + o(1) & \text{if } 0 \leq r \leq \alpha = 0.294; \\ \omega - (1 - r)\frac{\omega-2}{1-\alpha} & \text{if } \alpha < r \leq 1. \end{cases}$$

For every $0 \leq t \leq 1 \leq r$:

$$\omega(t,1,r) \leq \begin{cases} r+1+o(1) & 0 \leq t \leq \alpha; \\ r+1+(t-\alpha)\frac{\omega-2}{1-\alpha}+o(1) & \alpha < t \leq 1. \end{cases}$$

We have three cases, depending on the value of $k \pmod 3$.

Case 1: $k = 3h$**,** $h \in \mathbb{N}$**.** In this case $\mu_2 = \mu_3 = \mu$ where:

$$\omega(h,h,h-1+\mu) - \mu = \omega(h,h-1,h-1+\mu).$$

For $h = 1$ we have:

$$\omega - (1-\mu)\frac{\omega-2}{1-\alpha} - \mu = 1+\mu.$$

Then $\mu = \frac{1+\alpha-\alpha\omega}{4-2\alpha-\omega}$ and the update cost is $\mathrm{O}(n^{1+\mu}) = \mathrm{O}(n^{1.575})$. For $h \geq 2$ we have:

$$h\omega - (1-\mu)\frac{\omega-2}{1-\alpha} - \mu = \omega(h-1) + 1 + \mu\left(1-\alpha\frac{\omega-2}{1-\alpha}\right).$$

Then $\mu = \frac{1+\alpha-\alpha\omega}{4-\alpha\omega-\omega}$ and the update time is $\mathrm{O}(n^{h\omega-\omega+1+\mu(1-\alpha\frac{\omega-2}{1-\alpha})}) = \mathrm{O}(n^{h\omega-0.832})$.

Case 2: $k = 3h+1$**,** $h \in \mathbb{N}$**.** In this case:

$$\omega(h,h+1,h-1+\mu_2) - \mu_2 = \omega(h,h,h-1+\mu_2),$$

and

$$\omega(h,h,h+\mu_3) - \mu_3 = \omega(h,h-1,h+\mu_3).$$

For $h = 1$ we have:

$$3 - \mu_2 = \omega - (1-\mu_2)\frac{\omega-2}{1-\alpha},$$

and

$$\mu_3 + \omega - \mu_3 = 2 + \mu_3.$$

Then $\mu_2 = \frac{1-\alpha(3-\omega)}{\omega-1-\delta}$, $\mu_3 = \omega - 2$, and the update time is $\mathrm{O}(n^{\frac{\omega(3-\delta)-4}{\omega-1-\delta}} + n^{\omega}) = \mathrm{O}(n^{\omega})$. For $h \geq 2$ we have:

$$h\omega + 1 - (1-\mu_2)\frac{\omega-2}{1-\alpha} - \mu_2 = h\omega - (1-\mu_2)\frac{\omega-2}{1-\alpha},$$

and

$$\mu_3 + h\omega - \mu_3 = h\omega + \mu_3 - \alpha\frac{\omega - 2}{1 - \alpha}.$$

Then $\mu_2 = 1$, $\mu_3 = \alpha\frac{\omega-2}{1-\alpha}$, and the complexity is $\mathrm{O}(n^{h\omega})$.

Case 3: $k = 3h + 2$, $h \in \mathbb{N}$. In this case $\mu_2 = \mu_3 = \mu$ where:

$$\omega(h, h + 1, h + \mu) - \mu = \omega(h, h, h + \mu).$$

For any h we have:

$$h\omega + 1 + \mu\left(1 - \alpha\frac{\omega - 2}{1 - \alpha}\right) - \mu = \mu + h\omega.$$

Then $\mu = \frac{1-\alpha}{1+\alpha(\omega-3)}$ and the complexity is $\mathrm{O}(n^{h\omega+\mu}) = \mathrm{O}(n^{h\omega+0.865})$.

These results are summarized in Table 6 and 7. It follows that the update cost is

k	μ_2 and μ_3
3	$\mu_2 = \mu_3 = \frac{1+\alpha-\alpha\omega}{4-2\alpha-\omega}$
4	$\mu_2 = \frac{1-\alpha(3-\omega)}{\omega-1-\alpha}$; $\mu_3 = \omega - 2$
$3h + 2 \geq 5$	$\mu_2 = \mu_3 = \frac{1-\alpha}{1+\alpha(\omega-3)}$
$3h \geq 6$	$\mu_2 = \mu_3 = \frac{1+\alpha-\alpha\omega}{4-\alpha\omega-\omega}$
$3h + 1 \geq 7$	$\mu_2 = 1$; $\mu_3 = \alpha\frac{\omega-2}{1-\alpha}$

Table 6: Optimal values of μ_2 and μ_3 for varying k.

k	$\widetilde{\beta}(k)$
3	$\frac{5-\alpha-\omega(\alpha+1)}{4-2\alpha-\omega}$
$3h + 1 \geq 4$	$h\omega$
$3h + 2 \geq 5$	$h\omega + \frac{1-\alpha}{1+\alpha(\omega-3)}$
$3h \geq 6$	$h\omega - \omega + 1 + (\frac{1+\alpha-\alpha\omega}{4-\alpha\omega-\omega})(1 - \alpha\frac{\omega-2}{1-\alpha})$

Table 7: Values of $\widetilde{\beta}(k)$ for varying k.

$O(n^{\widetilde{\beta}(k)}) = O(n^{\beta(k)-\delta(k)})$, where:

$$\delta(k) = \begin{cases} 0.800 & \text{if } k = 3; \\ 0.832 & \text{if } k \pmod 3 = 0,\ k \geq 6; \\ 1.000 & \text{if } k \pmod 3 = 1; \\ 0.978 & \text{if } k \pmod 3 = 2. \end{cases}$$

Thus our algorithm performs updates roughly $n^{0.8}$ times faster than recomputing everything from scratch.

In Table 8, the complexity of the decremental algorithm is compared with the complexity of the static algorithm of Section 3.1 for $3 \leq k \leq 8$.

k	Static [Section 3.1]	Decremental
3	$O(n^{2.376})$	$O(n^{1.575})$
4	$O(n^{3.376})$	$O(n^{2.376})$
5	$O(n^{4.220})$	$O(n^{3.241})$
6	$O(n^{4.751})$	$O(n^{3.919})$
7	$O(n^{5.751})$	$O(n^{4.751})$
8	$O(n^{6.595})$	$O(n^{5.616})$

Table 8: Running time comparison of static and decremental algorithms for clique problem.

4.2 Inverse Consistency

Let G be the consistency graph corresponding to a binary constraints network $\mathcal{N}$. Consider the problem of computing the largest (for number of nodes) induced subgraph $G_{\ell\text{-IC}}$ of the consistency graph G such that all its partitions are non-empty and all its nodes are ℓ-inverse-consistent, or decide that such graph does not exist. This problem is well defined:

Lemma 4.2.1 *Let $\mathcal{G}_{\ell\text{-}IC}$ be the set of all the induced subgraphs of G such that all their partitions are non-empty and all their nodes are ℓ-inverse-consistent. If $\mathcal{G}_{\ell\text{-}IC}$ is not empty, it contains a unique graph $G_{\ell\text{-}IC}$ of maximum order.*

Proof. Suppose that there exist two distinct graphs $G_1 = G[V_1]$ and $G_2 = G[V_2]$ in $\mathcal{G}_{\ell\text{-IC}}$ of maximum order. Then $G' = G[V_1 \cup V_2]$ is an induced subgraph of G, of order strictly greater than G_1 and G_2, whose nodes are ℓ-inverse-consistent, which is a contradiction. □

The fastest algorithm known to solve this problem [20] has a $O(n^\ell d^\ell)$ time complexity. In this section we present a faster algorithm for the same problem, which is based on the decremental algorithm of Section 4.1. Its time complexity is $O(n^\ell d^{\widetilde{\beta}(\ell)+1})$. This improves on the $O(n^\ell d^\ell)$ bound for every $\ell \geq 3$.

The algorithm works as follows. It removes from G the nodes which are not ℓ-inverse-consistent one by one. When a node is removed, all the edges incident on it are also removed. The procedure ends when all the nodes in G are ℓ-inverse-consistent or a partition becomes empty. In the first case at the end of the procedure G is equal to $G_{\ell\text{-IC}}$. In the second case, $\mathcal{G}_{\ell\text{-IC}}$ is empty.

We have to show how nodes which are not ℓ-inverse-consistent are detected along the way. First of all, the algorithm checks all the nodes and remove the ones which are not ℓ-inverse-consistent. Then it propagates the effects of deletions. In fact, the deletion of one node can induce as a side effect the deletion of other nodes (which were previously recognized as ℓ-inverse-consistent). In particular, consider the deletion of a node (i, a). Let $\mathcal{G}_\ell(i)$ be the set of graphs induced on G by any possible choice of ℓ distinct partitions which include partition V_i. For each graph G' in $\mathcal{G}_\ell(i)$ and for each node (j, b) of G', one has to check whether (j, b) belongs to at least one ℓ-clique of G'. This can be efficiently checked via the decremental algorithm of Section 4.1.

In more details, the algorithm uses a set `DelSet` of integers to keep trace of the partitions into which a deletion occurred: whenever a node (i, a) is removed, i is stored in `DelSet`. We can distinguish two main steps in the algorithm: an initialization step and a propagation step. In the initialization step, for each node (i, a), the algorithm checks for each G' in $\mathcal{G}_\ell(i)$ whether (i, a) is contained in at least one ℓ-clique of G'. If this is not true, node (i, a) is removed from G and i is added to `DelSet`. Notice that i could be already contained in `DelSet`. In that case, `DelSet` is not modified.

In the propagation step, until `DelSet` is not empty, the algorithm extracts an integer j from `DelSet` and it executes the following steps. For each G' in $\mathcal{G}_\ell(j)$ and for each node (i,a) in G', $i \neq j$, it checks whether (i,a) is contained in at least one ℓ-clique of G'. If not, it removes (i,a) from G and it adds i to `DelSet`.

We have to show how the algorithm checks whether a node of a graph G' is contained in at least one ℓ-clique. The idea is to use the algorithm of Section 4.1. For each graph G' induced by ℓ distinct partitions, the algorithm maintains the number of ℓ-cliques in which each node of G' is contained. Whenever a node (i,a) is removed, the algorithm updates consequently these quantities for each graph G' in $\mathcal{G}_\ell(i)$.

Theorem 4.2.1 *The algorithm above computes $G_{\ell\text{-}IC}$ or determines that $\mathcal{G}_{\ell\text{-}IC}$ is empty in $O(n^\ell d^{\widetilde{\beta}(\ell)+1})$ time.*

Proof. The number of iterations of the propagation step is bounded by the number of nodes. Then the algorithm halts. An ℓ-inverse-consistent node is clearly never removed. Consider the non-trivial case that the algorithm halts when no partition is empty. To show correctness, we have to prove that all the remaining nodes in G are ℓ-inverse-consistent. Assume by contradiction that, when the algorithm halts, G contains a node (i,a) which is not ℓ-inverse-consistent. Since all the nodes which are not ℓ-inverse-consistent in the original graph are removed during the initialization step, (i,a) must be not ℓ-inverse-consistent because of the deletions which occurred during the initialization and/or the propagation step. Consider the sequence $(i^{(1)},a^{(1)}),(i^{(2)},a^{(2)})\dots(i^{(p)},a^{(p)})$ in which nodes are removed from the graph. Let q, $q \in \{1,2\dots p\}$, be the smallest index such that (i,a) is not ℓ-inverse-consistent in the graph $G[V^{(q)}]$ induced by $V^{(q)} = V\backslash\{(i^{(1)},a^{(1)}),(i^{(2)},a^{(2)})\dots(i^{(q)},a^{(q)})\}$. Notice that node $(i^{(q)},a^{(q)}) = (j,b)$ must belong to a graph G' which contains partition V_i, and thus node (i,a). After the deletion of node (j,b), j is inserted in `DelSet`. In one of the following steps, j is extracted from `DelSet` and all the nodes in any graph G' of $\mathcal{G}_\ell(j)$ are checked. In particular, node (i,a) is checked. Since in that iteration the set of nodes still in G is a

subset of $V^{(q)}$, the node (i, a) is recognized as a node which is not ℓ-inverse-consistent and it is thus removed, which is a contradiction. Thus the algorithm is correct.

The time complexity of the algorithm is bounded by the cost of maintaining the number of ℓ-cliques in which each node of each graph G' is contained. The number of such graphs is $O(n^\ell)$ (that is the number of ways we can select ℓ from n partitions), and each graph contains $O(d)$ nodes. Then the total initialization cost is $O(n^\ell d^{\beta(\ell)})$. Each graph G' is interested by at most $O(d)$ deletions. Then the total update cost is $O(n^\ell d^{\widetilde{\beta}(\ell)+1})$. Thus the time complexity of the algorithm is $O(n^\ell(d^{\beta(\ell)} + d^{\widetilde{\beta}(\ell)+1})) = O(n^\ell d^{\widetilde{\beta}(\ell)+1})$. □

In particular, this algorithm reduces the time complexity to *enforce* path inverse consistency from $O(n^3d^3)$ to $O(n^3d^{2.575})$. The performance of the algorithm above and of the previous best are compared in Table 9 for $3 \leq \ell \leq 8$.

ℓ	Previous best [20]	This section
3	$O(n^3d^3)$	$O(n^3d^{2.575})$
4	$O(n^4d^4)$	$O(n^4d^{3.376})$
5	$O(n^5d^5)$	$O(n^5d^{4.241})$
6	$O(n^6d^6)$	$O(n^6d^{4.919})$
7	$O(n^7d^7)$	$O(n^7d^{5.751})$
8	$O(n^8d^8)$	$O(n^8d^{6.616})$

Table 9: Time complexity comparison of inverse consistency based filtering algorithms.

4.3 Max-Restricted Path Consistency

Let G be the consistency graph corresponding to a binary constraints network $\mathcal{N}$. Consider the problem of computing the largest (for number of nodes) induced subgraph $G_{\text{max-RPC}}$ of the consistency graph G such that all its partitions are non-empty and all its nodes are max-restricted path consistent, or determines that such graph does not exist. This problem is well defined:

Lemma 4.3.1 *Let $\mathcal{G}_{max\text{-}RPC}$ be the set of all the induced subgraphs of G such that all their partitions are non-empty and all their nodes are max-restricted path consistent. If $\mathcal{G}_{max\text{-}RPC}$ is not empty, it contains a unique graph $G_{max\text{-}RPC}$ of maximum order.*

The proof of Lemma 4.3.1 is analogous to the proof of Lemma 4.2.1.

The fastest known max-restricted path consistency based filtering algorithm, `max-RPC-1` [21], has a O(end^3) time complexity and a O(end) space complexity. In Section 4.3.1 we present a new algorithm for the same task, which we denote by `max-RPC-2`, with the same time complexity as `max-RPC-1` but with a smaller space complexity, that is O(ed). In Section 4.3.2 we describe another max-restricted path consistency based filtering algorithm, which we denote by `max-RPC-3`, with a O($end^{2.575}$) time complexity and O(end^2) space complexity. This second algorithm makes use of a variant of the decremental algorithm described in Section 4.1.

4.3.1 Max-Restricted Path Consistency in Less Space

In this section we present a max-restricted path consistency based filtering algorithm, which we denote by `max-RPC-2`, with the same time complexity as `max-RPC-1`, which is O(end^3), but with a smaller space complexity, that is O(ed).

Let $G = (\{V_1, V_2 \dots V_n\}, E)$ be a consistency graph. Algorithm `max-RPC-3` removes non-max-restricted path consistent nodes from G one by one, until a partition becomes empty or all the remaining nodes are max-restricted path consistent.

The algorithm uses two kinds of data structures. A set `DelSet` of variables and a set $S_{(i,a)}$ of nodes for each node (i, a). The set `DelSet` is used to keep trace of the partitions from which a node has been removed. Whenever we remove a node (i, a), we store i in `DelSet`. The set $S_{(i,a)}$ is used to store the last path-consistent support (j, b) found for (i, a) on variable j, for any j linked to i.

The algorithm consists of two main steps: an initialization step and a propagation step. In the initialization step we consider each node (i, a) and we check if it is max-restricted

path consistent. If not, we remove (i, a) from G and we add i to `DelSet`. Otherwise we store the path-consistent supports found for (i, a) in $S_{(i,a)}$.

In the propagation step we have to propagate efficiently the effects of deletions. In fact, the deletion of one node can induce as a side effect the deletion of other nodes (which were previously recognized as max-restricted path consistent). There are substantially two kinds of such situations. The first case is when we delete the unique path-consistent support (j, b) for node (i, a) on variable j. The second is when we remove the unique witness (j, b) on variable j for the pair $\{(i, a), (k, c)\}$, where (k, c) is the unique path-consistent support for (i, a) on variable k. In both cases the node (i, a) is not max-restricted path consistent.

Thus in the propagation step, until `DelSet` is not empty, we extract a variable j from `DelSet` and we proceed as follows. We consider any node (i, a), with i linked to j, and we check if (i, a) is not max-restricted path consistent because of one of the two situations described above. In particular, we first check if the last path-consistent support (j, b) found for (i, a) on j (which is stored in $S_{(i,a)}$), still belongs to G. If not, we search for a new path-consistent support for (i, a) on j, and we update $S_{(i,a)}$ accordingly. If such path-consistent support does not exist, (i, a) is removed from the graph and i is added to `DelSet`. Then, if (i, a) has not been removed in the previous step, for any variable k linked to both i and j, we consider the last path-consistent support (k, c) found for (i, a) on k (which is stored in $S_{(i,a)}$). We check if the pair $\{(i, a), (k, c)\}$ has a witness on variable j (by simply considering all the candidate witnesses (j, b)). If not, we search for a new path-consistent support for (i, a) on k, and we update $S_{(i,a)}$ accordingly. If such path-consistent support does not exist, (i, a) is removed from the graph and i is added to `DelSet`. In both cases, when we search for a new path-consistent support, we consider the candidates according to a given (arbitrary) order. This way, we make sure that the same potential path-consistent support is checked at most once.

Theorem 4.3.1 *Algorithm* `max-RPC-2` *computes* $G_{max\text{-}RPC}$ *or determines that* $\mathcal{G}_{max\text{-}RPC}$ *is empty in* $\mathrm{O}(ed)$ *space and* $\mathrm{O}(end^3)$ *time.*

Proof. Consider first the correctness of `max-RPC-2`. No node is removed more than once. Since the number of nodes deleted is an upper bound on the number of iterations of the propagation step, the algorithm halts. A max-restricted path consistent node is clearly never removed. Consider the non-trivial case that the algorithm halts when no partition is empty. We have to prove that all the remaining nodes in G are max-restricted path consistent. Assume by contradiction that, when the algorithm halts, G contains a non-max-restricted path consistent node (i, a). Since all the non-max-restricted path consistent nodes in the original graph are removed during the initialization step, (i, a) must be non-max-restricted path consistent because of the deletions which occurred during the initialization and/or the propagation step. Consider the sequence $(i^{(1)}, a^{(1)}), (i^{(2)}, a^{(2)}) \dots (i^{(p)}, a^{(p)})$ in which nodes are removed from the network. Let q, $q \in \{1, 2 \dots p\}$, be the smallest index such that (i, a) is non-max-restricted path consistent in the graph graph $G[V^{(q)}]$ induced by $V^{(q)} = V \backslash \{(i^{(1)}, a^{(1)}), (i^{(2)}, a^{(2)}) \dots (i^{(q)}, a^{(q)})\}$. Notice that variable $i^{(q)}$ must be linked to i. When node $(i^{(q)}, a^{(q)}) = (j, b)$ is removed, j is inserted in `DelSet`. In one of the following steps, j is extracted from `DelSet` and all the nodes (k, c), with k linked to j, are checked. In particular, the node (i, a) is checked. Since in that iteration the current set of nodes of G is a subset of $V^{(q)}$, the node (i, a) is recognized as non-max-restricted path consistent and thus removed from G, which is a contradiction.

Let us now analyze the running time and space requirements of `max-RPC-2`. Without loss of generality, we assume $e = \Omega(n)$. Moreover we indicate with e_i the number of variables j linked to i. The set `DelSet` requires $\mathrm{O}(n)$ space. For each (i, a), the set $S_{(i,a)}$ takes $\mathrm{O}(e_i)$ space. Then the space complexity of `max-RPC-2` is $\mathrm{O}(n + \sum_{i=1}^{k} e_i d) = \mathrm{O}(ed)$.

The time complexity is bounded by the cost of searching for witnesses (which is required for both searching for new path-consistent supports and for checking previously detected ones). Searching for a witness costs $\mathrm{O}(d)$. Then checking a potential path-consistent support (j, b) for a node (i, a) on variable j costs $\mathrm{O}(e_j + e_i d) = \mathrm{O}(n + e_i d)$. Node (i, a) has $\mathrm{O}(d)$

potential path-consistent supports on each of the $O(e_i)$ variables j linked to i. No potential path-consistent support is checked more than once. Thus the total cost to search for new path-consistent supports is $O\left(\sum_{i=1}^{n} e_i d^2(n + e_i d)\right) = O(end^3)$. Whenever a deletion occurs into a domain D_j, we have to search for a witness on j for all the pairs of nodes $\{(i, a), (k, c)\}$, where (k, c) is the current path-consistent support for (i, a) on k and i, j and k are pairwise linked. The number of such pairs is $O(e_j^2 d)$, and we can detect them in $O((e_i + e_j)d) = O((n + e_j)d)$ steps. Each check costs $O(d)$. Since domain D_j is interested by $O(d)$ deletions, the total cost of these checks is $O\left(\sum_{j=1}^{n} d^2(nd + e_j^2 d)\right) = O(end^3)$. Thus the time complexity of `max-RPC-2` is $O(end^3)$. □

Notice that Algorithm `max-RPC-2` may check the same potential witness for a given pair of nodes more than once. Since these redundant checks are relatively infrequent, they do not affect the total time complexity. Algorithm `max-RPC-1` instead, avoids redundancies by storing (in $O(end)$ space) all the witnesses found: this way, the space complexity is increased without reducing asymptotically the time complexity.

Experiments We compared Algorithms `max-RPC-1` and `max-RPC-2` on *random consistency graphs*. A random consistency graph [8] is generated as follows. Given the number n of variables and the common cardinality d of the domains, two distinct variables are linked with probability p_d (*constraints density*). Two nodes (i, a) and (j, b), where i and j are linked, are not adjacent with probability p_t (*constraints tightness*).

Both algorithms were implemented in `C++` with the same programming style and tested on a Pentium III-500 MHz, with 512 KB L2 Cache and 256 MB of RAM memory, under Linux. In Figure 4.1 the CPU-time performance of the two algorithms are compared in the case $k = 200$ and $d = 30$, for two different values of the constraints density. The CPU-time values were obtained by averaging over 100 random instances.

For both densities and algorithms, the CPU-time increases with the tightness until a pick

is reached, and then decreases sharply. For low values of tightness, the number of deletions is low. Then there are few iterations in the propagation step and the total cost is dominated by the initialization cost. When the tightness is high, many nodes are deleted during the initialization step. Then one of the partitions becomes empty soon and both algorithms halt quickly. As it can be inferred from Figure 4.1, Algorithm `max-RPC-2` is faster than `max-RPC-1` for low and high values of tightness. The reason for this behavior could be that the initialization cost, which dominates the total cost for values of tightness far-away from the pick, is smaller in `max-RPC-2`. Around the pick the performance of `max-RPC-2` degrades relatively faster than the performance of `max-RPC-1`. In particular, for density $p_d = 0.15$, `max-RPC-1` is faster than `max-RPC-2` for values of tightness around $p_t = 0.7$. The reason for this could be that the cost of the propagation step, which is not negligible around the pick, is higher for `max-RPC-2` because of the redundant checks. Interestingly, Algorithm `max-RPC-2` is always faster than `max-RPC-1` for low values of density. In fact, the consistency graphs derived from the applications often have low densities.

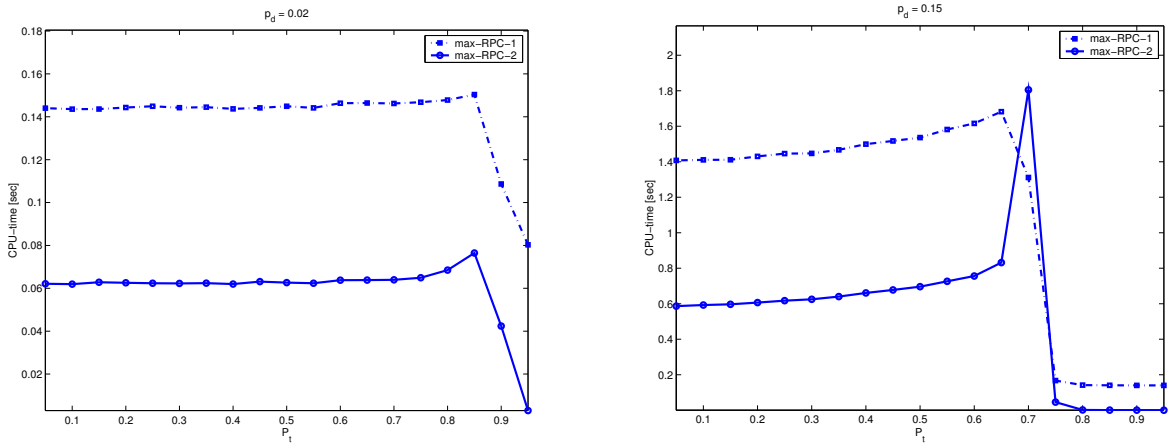


Figure 4.1: Comparison of Algorithms `max-RPC-1` and `max-RPC-2` on random consistency graphs, with $n = 200$ and $d = 30$.

4.3.2 Max-Restricted Path Consistency in Less Time

In this section we describe a variant of Algorithm `max-RPC-2`, denoted by `max-RPC-3`, which is asymptotically faster than both `max-RPC-1` and `max-RPC-2`, but which has a higher space complexity. In Section 4.2 we described a fast ℓ-inverse consistency based filtering algorithm, based on the decremental algorithm of Section 4.1. In this section we use a similar approach.

Algorithm `max-RPC-3` has the same basic structure of `max-RPC-2`. The main difference is the way it checks whether a pair $\{(i,a),(j,b)\}$ has a witness on a variable k. Let us define an integer matrix $W_{i,k,j}$ such that $W_{i,k,j}[a,b]$ is equal to the number of witnesses for the pair $\{(i,a),(j,b)\}$ on variable k. Clearly $W_{i,k,j}[a,b] = 0$ if $A_{i,j}[a,b] = 0$, and it is equal to $(A_{i,k} \cdot A_{k,j})[a,b] = A_{i,k,j}[a,b]$ otherwise:

$$W_{i,k,j}[a_i,a_j] = A_{i,k,j}[a_i,a_j] \cdot A_{i,j}[a_i,a_j]. \tag{4.2}$$

Note that matrix $A_{i,k,j}$ can be computed, via fast matrix multiplication, in $O(d^\omega)$ steps, $\omega < 2.376$ [17]. A simple-minded approach could then be to maintain an updated version of matrix $A_{i,k,j}$, for any triple $\{i,k,j\}$ of pairwise linked variables. This allows to know the number of witnesses for a given pair in $O(1)$ time (*query time*). When we remove a value assignment (k,c), the cost of updating $A_{i,k,j}$ is $O(d^2)$ (*update time*).

We will now show how to reduce the update time at the cost of increasing the query time. Instead of maintaining $A_{i,k,j}$, we maintain a lazy version $A'_{i,k,j}$ of it. Whenever we remove a value assignment (k,c), instead of updating $A'_{i,k,j}$, we store c in a set $D_{i,k,j}$. When the cardinality of $D_{i,k,j}$ reaches a given threshold n^μ, $\mu \in [0,1]$, we update $A'_{i,k,j}$ according to the following relation:

$$\forall a \in D_i, b \in D_j : A'_{i,k,j}[a,b] = A'_{i,k,j}[a,b] - A_{i,j}[a,b] \sum_{c \in D_{i,k,j}} A_{i,k}[a,c] \cdot A_{k,j}[c,b],$$

and then we empty $D_{i,k,j}$. The update time is then $O(d^{\omega(1,\mu,1)})$, where the time complexity of multiplying a $d^r \times d^s$ matrix by a $d^s \times d^t$ matrix is denoted by $O(d^{\omega(r,s,t)})$. Since we update $A'_{i,k,j}$ every $\Theta(d^\mu)$ deletions, the amortized update time per deletion is $O(d^{\omega(1,\mu,1)-\mu})$.

Clearly, when we need to compute $W_{i,k,j}[a,b]$, we cannot apply equation (4.2) directly, but we have to consider the deleted values in $D_{i,k,j}$ also:

$$W_{i,k,j}[a,b] = A_{i,j}[a,b] \cdot \left(A'_{i,k,j}[a,b] - \sum_{c \in D_{i,k,j}} A_{i,k}[a,c] \cdot A_{k,j}[c,b] \right).$$

Since $D_{i,k,j}$ contains at most d^μ elements, the query time is $\mathrm{O}(d^\mu)$.

Notice that, by setting μ to zero, we obtain the same performance as the simple-minded algorithm, that is a $\mathrm{O}(1)$ query time and a $\mathrm{O}(n^2)$ update time. For a reason that will be clearer from the analysis, we fix μ such that $1 + 2\mu = \omega(1, \mu, 1)$. The current best bounds on $\omega(r, s, t)$ [16, 43] imply $\mu < 0.575$.

Theorem 4.3.2 *Algorithm* `max-RPC-3` *computes* $G_{max\text{-}RPC}$ *or determines that* $\mathcal{G}_{max\text{-}RPC}$ *is empty in* $\mathrm{O}(end^2)$ *space and* $\mathrm{O}(end^{2+\mu}) = \mathrm{O}(end^{2.575})$ *time.*

Proof. The correctness of `max-RPC-3` follows directly from the correctness of `max-RPC-2` and of the procedure described above to update the number of witnesses.

Let us now consider the complexity of the algorithm. As for `max-RPC-2`, the space required to store the set `DelSet` and the sets $S_{(i,a)}$ is $\mathrm{O}(ed)$. Matrices $A'_{i,k,j}$ and sets $D_{i,k,j}$ take $\mathrm{O}(d^2 \sum_{i=1}^{n} e_i^2) = \mathrm{O}(end^2)$ space. Then the space complexity of `max-RPC-3` is $\mathrm{O}(end^2)$.

The time complexity of the algorithm is bounded by the cost of initializing and updating the matrices $A'_{i,k,j}$, and of making queries. Each $A'_{i,k,j}$ can be initialized in $\mathrm{O}(d^\omega)$ steps, with a total $\mathrm{O}(end^\omega)$ initialization cost. Each $A'_{i,k,j}$ is updated $\mathrm{O}(d)$ times, with a total $\mathrm{O}(end^{1+\omega(1,\mu,1)-\mu}) = \mathrm{O}(end^{2+\mu})$ updating cost. From the analysis of `max-RPC-2`, we check whether a pair of nodes has a witness on a given variable $\mathrm{O}(end^2)$ times. Each check costs $\mathrm{O}(d^\mu)$. Then the total query cost is $\mathrm{O}(end^{2+\mu})$. Thus the time complexity of `max-RPC-3` is $\mathrm{O}(en(d^\omega + d^{2+\mu})) = \mathrm{O}(end^{2+\mu}) = \mathrm{O}(end^{2.575})$. □

Fast matrix multiplication algorithms seem to be faster in practice than classical ones only for matrices of relatively high dimension. For this reason and for its high space complexity, Algorithm `max-RPC-3` seems to be mainly of theoretical interest.

Chapter 5

Dominating Set and Set Cover

A *dominating set* of an undirected graph $G = (V, E)$ of n nodes is a subset V' of V such that all the nodes which do not belong to V' are adjacent to at least one node in V'. The *dominating set* problem consists in determining whether G admits a dominating set of cardinality k. The *minimum dominating set* problem consists in determining the minimum cardinality of a dominating set of G. In Section 5.1 we provide an improved algorithm for dominating set for any fixed $k \geq 2$, which is based on fast matrix multiplication. The running time of our algorithm is $O(n^{\omega(\lfloor k/2 \rfloor, 1, \lceil k/2 \rceil)}) = O(n^{k+\omega-2})$, where $\omega < 2.376$ is fast matrix multiplication exponent. This answers a question posed by Regan [70], who asked whether there exists an algorithm which is faster than the $O(n^{k+1})$ trivial algorithm.

Let $\mathcal{S}$ be a collection of subsets of a given *universe* $\mathcal{U}$. A *set cover* of $\mathcal{S}$ is a subset $\mathcal{S}'$ of $\mathcal{S}$ which *covers* all the elements of $\mathcal{U}$:

$$\mathcal{U} = \cup_{S \in \mathcal{S}'} S.$$

The *minimum set cover problem* consists in determining the minimum cardinality $msc(\mathcal{S})$ of a set cover of $\mathcal{S}$. In Section 5.2 we present an algorithm which solves minimum set cover in $O(1.3424^d)$ time, where $d = |\mathcal{S}| + |\mathcal{U}|$ is the *dimension* of the problem. Minimum dominating set can be formulated as a minimum set cover problem of dimension $d = 2n$. It follows that minimum dominating set can be solved in $O(1.3424^{2n}) = O(1.8021^n)$ time. We remark that

the previous best algorithm for minimum dominating set is the trivial $O(2^n n^2)$ algorithm which enumerates and checks all the subsets of nodes.

5.1 Small Dominating Sets

In this section we present a $O(n^{\omega(\lfloor k/2 \rfloor, 1, \lceil k/2 \rceil)}) = O(n^{k+\omega-2})$ algorithm for dominating set.

Let V_h denote the set of subsets of V of cardinality h, $h \in \mathbb{N}$. Let moreover D_h be a 0-1 matrix whose rows and columns are indexed by the elements of V_h and V respectively and such that, for any $w \in V_h$ and $v \in V$, $D_h[w, v] = 0$ if and only if w dominates v (w *dominates* v if v belongs to w or v is adjacent to at least one node in w).

Our algorithm for dominating set works as follows. It first computes the matrices D_{k_1} and D_{k_2}, where $k_1 = \lfloor k/2 \rfloor$ and $k_2 = \lceil k/2 \rceil$ (notice that $k = k_1 + k_2$). Then it computes the matrix $D' = D_{k_1} \cdot D_{k_2}^T$. Eventually the algorithm answers *yes* if D' contains a 0-entry and *no* otherwise.

Theorem 5.1.1 *The algorithm above determines whether an undirected graph G of n nodes admits a dominating set of cardinality k in $O(n^{\omega(\lfloor k/2 \rfloor, 1, \lceil k/2 \rceil)})$ time.*

Proof. Given $w \in V_{k_1}$ and $z \in V_{k_2}$, $D'[w, z]$ is the number of elements of V which are not dominated by $(w \cup z)$. This implies that G admits a dominating set of cardinality k (or smaller) if and only if D' contains at least one 0-entry. Thus the algorithm is correct.

Matrices D_{k_1} and D_{k_2} can be computed in linear time (in their size). Matrix D' can be computed in $O(n^{\omega(k_1,1,k_2)})$ time. This ends the proof. □

The algorithm above improves on the trivial algorithm for every value of $k \geq 2$ even if D' is computed via the straightforward decomposition in square blocks and fast square matrix multiplication:

$$\omega(\lfloor k/2 \rfloor, 1, \lceil k/2 \rceil) \leq \lfloor k/2 \rfloor + \lceil k/2 \rceil + 1 + (\omega - 3) \min\{\lfloor k/2 \rfloor, 1, \lceil k/2 \rceil\} = k + \omega - 2 < k + 1.$$

Better time bounds are obtained by using more sophisticated rectangular matrix multiplication algorithms [16, 43]. The complexities obtained by using fast rectangular matrix

multiplication are shown in Table 10 for $2 \leq k \leq 7$.

k	Previous best	This section
2	$O(n^3)$	$O(n^{2.376})$
3	$O(n^4)$	$O(n^{3.334})$
4	$O(n^5)$	$O(n^{4.220})$
5	$O(n^6)$	$O(n^{5.220})$
6	$O(n^7)$	$O(n^{6.063})$
7	$O(n^8)$	$O(n^{7.063})$

Table 10: Running time comparison for dominating set.

Interestingly, the complexity of our algorithm is $O(n^{k+o(1)})$ for any fixed $k \geq 8$. In fact, the following bound holds [16, 43]:

$$\omega(1,1,r) \leq \begin{cases} 2+o(1) & \text{if } 0 \leq r \leq \delta = 0.294, \\ \omega + \frac{(1-r)(2-\omega)}{1-\delta} & \text{if } \delta < r \leq 1. \end{cases}$$

This implies that, for any fixed $k \geq 8$:

$$\begin{aligned} \omega(\lfloor k/2 \rfloor, 1, \lceil k/2 \rceil) &\leq \lceil k/2 \rceil - \lfloor k/2 \rfloor + \omega(\lfloor k/2 \rfloor, 1, \lfloor k/2 \rfloor) \\ &\leq \lceil k/2 \rceil - \lfloor k/2 \rfloor + \lfloor k/2 \rfloor (2 + o(1)) \\ &= k + o(1). \end{aligned}$$

5.2 Minimum Dominating Set

Let $\mathcal{S}$ be a collection of subsets of a given *universe* $\mathcal{U}$. Assume that $\mathcal{S}$ *covers* $\mathcal{U}$:

$$\bigcup_{S \in \mathcal{S}} S = \mathcal{U}.$$

The *minimum set cover* problem consists in determining the minimum cardinality $msc(\mathcal{S})$ of a set cover of $\mathcal{S}$. Without loss of generality, we can assume that $\mathcal{S}$ does not contain the empty set (since it does not belong to any minimum set cover). In this section we present an algorithm which solves minimum set cover in $O(1.3424^d)$ time, where $d = |\mathcal{S}| + |\mathcal{U}|$ is the *dimension* of the problem. Minimum dominating set can be formulated as a minimum set

cover problem where there is a set $N[v] = N(v) \cup \{v\}$ for each node v. The dimension of the minimum set cover formulation of minimum dominating set is $d = 2n$, where n is the number of nodes in the graph. It follows that minimum dominating set can be solved in $\mathrm{O}(1.3424^{2n}) = \mathrm{O}(1.8021^n)$ time.

The rest of this section is organized as follows. In Section 5.2.1 we describe a $\mathrm{O}(1.3803^d)$ polynomial-space recursive algorithm for minimum set cover. In Section 5.2.2 we show how to derive a $\mathrm{O}(1.3424^d)$ exponential-space recursive algorithm via dynamic programming.

5.2.1 A Polynomial-Space Algorithm

In this section we describe a $\mathrm{O}(1.3803^d)$ polynomial-space recursive algorithm `MSC` for minimum set cover, where d is the dimension of the problem. It follows that minimum dominating set can be solved in $\mathrm{O}(1.3803^{2n}) = \mathrm{O}(1.9053^n)$ time.

By $del(\mathcal{S}, R)$ we denote the collection which is obtained from $\mathcal{S}$ by removing the elements of R from each S in $\mathcal{S}$, and by removing the empty sets obtained (if any):

$$del(\mathcal{S}, R) = \{S' \neq \emptyset : S' = S \backslash R, S \in \mathcal{S}\}.$$

The algorithm is based on the following simple properties of set covers.

Lemma 5.2.1 *Let $\mathcal{S}$ be an instance of minimum set cover. The following properties hold:*

1. *If there is a set S in $\mathcal{S}$ which is (properly) included in another set R in $\mathcal{S}$ ($S \subset R$), then there is a minimum set cover which does not contain S. In particular:*

$$msc(\mathcal{S}) = msc(\mathcal{S} \backslash \{S\}).$$

2. *If there is an element s of $\mathcal{U}$ which belongs to a unique $S \in \mathcal{S}$, then S belongs to every set cover. In particular:*

$$msc(\mathcal{S}) = 1 + msc(del(\mathcal{S}, S)).$$

3. *For all $S \in \mathcal{S}$, the following holds:*

$$msc(\mathcal{S}) = \min\{msc(\mathcal{S} \backslash \{S\}), 1 + msc(del(\mathcal{S}, S))\}.$$

Figure 19 Recursive algorithm for minimum set cover.

```
1    int MSC(S) {
2        if(|S| = 0) return 0;
3        if(∃S, R ∈ S : S ⊂ R) return MSC(S\{S});
4        if(∃s ∈ U∃ a unique S ∈ S : s ∈ S) return 1 + MSC(del(S, S));
5        take S ∈ S of maximum cardinality;
6        return min{MSC(S\{S}), 1 + MSC(del(S, S))};
7    }
```

Note that the sets of cardinality one satisfy exactly one of the Properties (1) and (2).

A basic version of the algorithm is described in Figure 19. The base case (line 2) is when $|\mathcal{S}| = 0$. In that case, $msc(\mathcal{S}) = 0$. Otherwise (lines 3 and 4), the algorithm tries to reduce the dimension of the problem without branching, by applying one of the Properties (1) and (2). If none of the two properties above applies, the algorithm simply takes (line 5) a set $S \in \mathcal{S}$ of maximum cardinality and branches (line 6) according to Property (3).

Theorem 5.2.1 *Algorithm* `MSC` *solves minimum set cover in* $O(1.3803^d)$ *time, where* d *is the dimension of the problem.*

Proof. The correctness of the algorithm is a straightforward consequence of Lemma 5.2.1.

Let $N_h(d)$ denote the number of subproblems of dimension h solved by the algorithm to solve a problem of dimension d. Clearly, $N_h(d) = 0$ for $h > d$ (the subproblems are of dimension lower than the dimension of the original problem). Moreover, $N_d(d) = 1$ (considering the original problem as one of the subproblems). Consider the case $h < d$ (which implies $|\mathcal{S}| \neq 0$). If one of the conditions of lines 3 and 4 is satisfied, the algorithm generates a unique subproblem of dimension at most $d - 1$. Thus:

$$N_h(d) \leq N_h(d-1).$$

Otherwise, the algorithm takes a set S of maximum cardinality ($|S| \geq 2$), and it branches on the two subproblems $\mathcal{S}_1 = \mathcal{S}\backslash\{S\}$ and $\mathcal{S}_2 = del(\mathcal{S}, S)$. The dimension of $\mathcal{S}_1$ is $d-1$ (one set is removed from $\mathcal{S}$). If $|S| \geq 3$, the dimension of $\mathcal{S}_2$ is at most $d-4$ (one set is removed

from $\mathcal{S}$ and at least three elements are removed from $\mathcal{U}$). Thus:

$$N_h(d) \leq N_h(d-1) + N_h(d-4).$$

Otherwise (all the sets in $\mathcal{S}$ have cardinality two), the dimension of $\mathcal{S}_2$ is $d-3$ (one set is removed from $\mathcal{S}$ and two elements are removed from $\mathcal{U}$). Moreover, $\mathcal{S}_2$ must contain one set S' of cardinality one. Then, while solving the subproblem $\mathcal{S}_2$, one of the conditions of lines 3 and 4 is satisfied. Thus:

$$N_h(d) \leq N_h(d-1) + N_h(d-3-1) = N_h(d-1) + N_h(d-4).$$

A valid upper bound on $N_h(d)$ is $N_h(d) \leq c^{d-h}$, where $c = 1.3802\ldots < 1.3803$ is the (unique) positive root of the polynomial $(x^4 - x^3 - 1)$. This implies that the total number $N(d)$ of subproblems solved is:

$$N(d) = \sum_{h=0}^{d} N_h(d) \leq \sum_{h=0}^{d} c^{d-h} = \mathrm{O}(c^d).$$

The cost of solving a problem of dimension $h \leq d$, excluding the cost of solving the corresponding subproblems (if any), is upper bounded by a polynomial $p(d)$ of d. It follows that the time complexity of the algorithm is $\mathrm{O}(c^d p(d)) = \mathrm{O}(1.3803^d)$. □

Corollary 5.2.1 *There is an algorithm which solves minimum dominating set in* $\mathrm{O}(1.9053^n)$ *time, where* n *is the number of nodes in the graph.*

Proof. It is sufficient to reduce minimum dominating set to minimum set cover and then use algorithm `MSC`. The time complexity of this algorithm is $\mathrm{O}(1.3803^{2n}) = \mathrm{O}(1.9053^n)$. □

5.2.2 An Exponential-Space Algorithm

In this section we show how to reduce the time complexity of the algorithm for minimum set cover of Section 5.2.1 via *dynamic programming*, at the cost of an exponential space complexity. In particular, we present a $\mathrm{O}(1.3424^d)$ exponential-space recursive algorithm

`MSC'` for minimum set cover. It follows that dominating set can be solved in $O(1.3424^{2n}) = O(1.8021^n)$ time.

The technique is similar to the one used by Robson in the context of maximum independent set [72]. While solving a problem $\mathcal{S}$, the same subproblem $\mathcal{S}'$ can appear many times. The idea is then to store the solutions of all the subproblems solved in a database. Whenever a new subproblem is generated, this database is checked to verify whether the solution of that subproblem is already available. This way, one ensures that a given subproblem is solved at most once. The database can be implemented in such a way that the *query time* is logarithmic in the number of solutions stored.

Theorem 5.2.2 *Algorithm* `MSC'` *solves minimum set cover in* $O(1.3424^d)$ *time, where* d *is the dimension of the problem.*

Proof. The correctness of the algorithm follows from the correctness of algorithm `MSC`.

Let $N_h(d)$ denote the number of subproblems of dimension $h \in \{0, 1 \ldots d\}$ solved by the algorithm to solve a problem of dimension d. From the proof of Theorem 5.2.1, $N_h(d) \leq c^{d-h}$, where $c = 1.3802\ldots < 1.3803$ is the positive root of the polynomial $(x^4 - x^3 - 1)$. Each subproblem $\mathcal{S}'$ is of the form:

$$\mathcal{S}' = del(\mathcal{S} \backslash \mathcal{S}^*, \mathcal{U}^*),$$

where $\mathcal{S}^* \subseteq \mathcal{S}$ and $\mathcal{U}^* \subseteq \mathcal{U}$. This implies that $N_h(d) \leq \binom{d}{h}$. Let h' be the largest integer in $\{0, 1 \ldots \lfloor d/2 \rfloor\}$ such that $\binom{d}{h'} < c^{d-h'}$. The total number $N(d)$ of subproblems solved is:

$$N(d) = \sum_{h=0}^{d} N_h(d) \leq \sum_{h=0}^{h'} \binom{d}{h} + \sum_{h=h'+1}^{d} c^{d-h} = O(\binom{d}{h'} + c^{d-h'}) = O(c^{d-h'}).$$

It follows from Stirling's approximation that $N(d)$ is $O(c^{(1-\alpha)d})$, where α satisfies:

$$c^{1-\alpha} = \frac{1}{\alpha^\alpha (1-\alpha)^{1-\alpha}}.$$

The cost of each query to the database is polynomial in d. Thus the cost of solving a problem of dimension $h \leq d$, excluding the cost of solving the corresponding subproblems (if any),

is upper bounded by a polynomial $p(d)$ of d. It follows that the time complexity of the algorithm is $O(c^{(1-\alpha)d}p(d)) = O(1.3424^d)$. □

Corollary 5.2.2 *There is an algorithm which solves minimum dominating set in* $O(1.8021^n)$ *time, where* n *is the number of nodes in the graph.*

Proof. It is sufficient to reduce minimum dominating set to minimum set cover and then use algorithm `MSC'`. The time complexity of this algorithm is $O(1.3424^{2n}) = O(1.8021^n)$. □

5.2.3 Remarks

The main result of this section is a $O(1.8021^n)$ algorithm for minimum dominating set. We remark that the previous (asymptotically) fastest algorithm for this problem is the trivial $\Omega(2^n)$ algorithm which enumerates and checks all the subsets of nodes.

We believe that the time complexity of the algorithm presented can be reduced via a refined use of case analysis and dynamic programming. This deserves further investigation.

Conclusions

A lot of effort has been devoted in last decades to develop asymptotically faster (exponential-time) exact algorithms for NP-complete and NP-hard problems, such as *maximum independent set* [5, 45, 72, 73], *vertex cover* [3, 14, 66], *(maximum) satisfiability* [4, 19, 64, 67], 3-*coloring* [6, 31] and many others.

This thesis has been focused on three fundamental NP-complete graph problems: *vertex cover*, *independent set* and *dominating set*. We presented asymptotically improved algorithms for the detection of vertex covers, independent sets and dominating sets of size "sufficiently" smaller than the number of nodes. We moreover delivered the first non-trivial asymptotic bound on minimum dominating set, thus significatively extending the class of NP-hard problems for which non-trivial asymptotic bounds are known.

There are several open problems which we would like to investigate. Consider for example *minimum set cover*. We have already shown that it can be solved in $O(1.35^d)$ time, where $d = n+m$ is the sum of the n sets available and of the m elements which need to be covered. This means an improvement on the trivial $\Omega(2^n)$ algorithm when m is not too large (for example, when $m \leq n$). Is it possible to improve on the trivial algorithm for arbitrary values of m?

A maybe even more significative open problem concerns *integer linear programming*. Even in the case the n integer variables can take value zero and one only, no algorithm is known which is faster than the trivial $\Omega(2^n)$ enumerative one. Showing that a faster algorithm exists could be extremely interesting for both theoretical and practical reasons.

Another topic which we would like to investigate further is cycle detection and, in particular, triangle detection. The fastest triangle-detection algorithms known have time complexity $O(n^\omega)$ and $O(e^{\frac{2\omega}{\omega+1}})$ in dense and sparse graphs respectively, where ω is square matrix multiplication exponent. It is conjectured that $\omega = 2$. This would imply that a triangle can be detected in linear time in dense graphs. Interestingly, this is not the case for sparse graphs, for which the time complexity is $O(e^{4/3})$ even if $\omega = 2$. Is it possible to detect a triangle in $O(e^{\omega/2})$ time? This seems to be a very hard and interesting open problem.

Acknowledgments

I wish to thank several persons for their invaluable support in the three years of my PhD.

First of all, I wish to thank my supervisor and coauthor Prof. Giuseppe F. Italiano, who helped me to move the first difficult steps in the research world.

I want also to thank my other two coauthors: Dr. Sunil L. Chandran and Dr. Friedrich (Fritz) Eisenbrand. I feel fortunate to have had the opportunity to work with them. In particular, I wish to thank Fritz, who was much more than a colleague for me, a friend, in the thirteen months I spent in the Max-Planck-Institut für Informatik of Saarbrücken. Most of the results of this thesis would not have been possible without his professional and moral support.

I take the opportunity to thank the other persons who made my long stay in Saarbrücken so pleasant, and in particular Prof. Kurt Mehlhorn, who gave me the opportunity of joining his research group in the first nine months at the Max-Planck-Institut. I wish also to mention, in alphabetic order, Dr. Hannah (yes, Hannah) Bast, Markus Behle, Niko Beerenwinkel, Dr. Dimitris Fotakis, Dr. Stefan Funke, Prof. Naveen Garg, Dr. Andreas Kämper, Dr. Spyros Kontogiannis, Dr. Piotr Krysta, Lars Kunert, Babak Mougouie, Dr. Venkatesh Srinivasan, Dr. Kavitha Telikepalli and Sven Thiel.

I want to thank all the staff of "Dipartimento di Informatica, Sistemi e Produzione", of the University of Rome "Tor Vergata". A special thank goes to the group of "Control Engineering", who helped me in the first months of the PhD. In particular, I wish to thank Prof. Salvatore Nicosia, who supported me in starting my PhD. I wish also to mention

Dr. Sergio Galeani, Prof. Osvaldo Maria Grasselli, Dr. Agostino Martinelli, Dr. Francesco Martinelli, Prof. Laura Menini, Prof. Antonio Tornambè, Prof. Paolo Valigi and Dr. Luca Zaccarian. I would like also to thank Prof. Daniel Pierre Bovet, Dr. Marco Cesati, Dr. Irene Finocchi, Dr. Gianpaolo Oriolo and Dr. Andrea Pacifici. A special mention goes to Dr. Gianpaolo Oriolo, for his valuable professional and moral support in the last months of my PhD.

There are several other people I would like to thank. In particular, Dr. Claudio Gentile, Prof. Stefano Leonardi, Prof. Alessandro Panconesi and Dr. Paolo Ventura.

Eventually, I wish to thank my loved ones, who give meaning to my entire life. A warm thank to my parents, who supported me morally and economically, to my brother "Stefo" and to my friends. The last thank goes to my "little star" Elisa, whom this thesis is dedicated to. Her lovely presence made the years of my PhD the most beautiful period in my life.

Bibliography

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the Association for Computing Machinery*, 42(4):844–856, 1995.

[2] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[3] R. Balasubramanian, M. Fellows, and V. Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65:163–168, 1998.

[4] N. Bansal and V. Raman. Upper bounds for MaxSat: further improved. In *International Symposium on Algorithms and Computation*, 1999.

[5] R. Beigel. Finding maximum independent sets in sparse and general graphs. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 856–857, 1999.

[6] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3446^n)$: a no-MIS algorithm. In *IEEE Symposium on Foundations of Computer Science*, pages 444–452, 1995.

[7] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Principles and Practice of Constraint Programming*, pages 52–66, 2000.

[8] C. Bessière. Arc-consistency and arc-consistency again. In *National Conference on Artificial Intelligence*, pages 179–190, 1994.

[9] B. Bollobás. *Extremal graph theory*, volume 11 of *London Mathematical Society Monographs*. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1978.

[10] J. Bondy and M. Simonovits. Cycles of even length in graphs. *Journal of Combinatorial Theory*, 16:97–105, 1974.

[11] P. Bürgisser, M. Clausen, and M. Shokrollahi. *Algebraic Complexity Theory*. Springer Verlag, 1997.

[12] J. F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM Journal on Computing*, 22(3):560–572, 1993.

[13] L. S. Chandran and F. Grandoni. Refined memorisation for vertex cover. Submitted for publication to "Information Processing Letters", 2003.

[14] J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[15] S. Cook. The complexity of theorem proving procedures. In *ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[16] D. Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13:42–49, 1997.

[17] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[19] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

[20] R. Debruyne. A property of path inverse consistency leading to an optimal PIC algorithm. In *European Conference on Artificial Intelligence*, pages 88–92, 2000.

[21] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Principles and Practice of Constraint Programming*, pages 312–326, 1997.

[22] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, (14):205–230, May 2001.

[23] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the O(n^2) barrier. In *IEEE Symposium on Foundations of Computer Science*, pages 381–389, 2000.

[24] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *IEEE Symposium on Foundations of Computer Science*, pages 260–267, 2001.

[25] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness. II. On completeness for $W[1]$. *Theoretical Computer Science*, 141(1-2):109–131, 1995.

[26] R. G. Downey and M. R. Fellows. *Parameterized complexity*. Monographs in Computer Science. Springer-Verlag, 1999.

[27] R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In J. K. F. Roberts and J. Nesetril, editors, *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 49–99, 1999.

[28] F. Eisenbrand and F. Grandoni. Detecting directed 4-cycles still faster. *Information Processing Letters*, 87(1):13–15, 2003.

[29] F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. 2003. To appear in "Theoretical Computer Science".

[30] C. D. Elfe and E. C. Freuder. Neighborhood inverse consistency preprocessing. In *National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, volume 1, pages 202–208, 1996.

[31] D. Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 329–337, 2001.

[32] P. Erdős. On the number of complete subgraphs contained in certain graphs. *Magyar Tudomanyos Akademia Matematikai Kutató Intezetenek Közlemenyei*, 7:459–464, 1962.

[33] M. R. Fellows. On the complexity of vertex set problems. Technical report, Computer Science Department, University of New Mexico, 1988.

[34] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *European Symposium on Algorithms*, pages 320–331, 1998.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freemann, 1979.

[36] F. Glover. Maximum matching in convex bipartite graph. *Naval Research Logistic Quarterly*, 14:313–316, 1967.

[37] F. Grandoni. A note on the complexity of minimum dominating set. Submitted for publication to "Journal of Algorithms", 2003.

[38] F. Grandoni and G. F. Italiano. Improved algorithms for max-restricted path consistency. In *Principles and Practice of Constraint Programming*, pages 858–862, 2003.

[39] N. B. Guernalec and A. Colmerauer. Narrowing a $2n$-block of sortings in $O(n \log n)$. In *Principles and Practice of Constraint Programming*, pages 2–16, 1997.

[40] E. Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 329–337, 2000.

[41] J. Håstad. Some optimal inapproximability results. In *ACM Symposium on the Theory of Computing*, pages 1–10, 1997.

[42] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182(1):105–142, 1998.

[43] X. Huang and V. Pan. Fast rectangular matrix multiplication and applications. *Journal of Complexity*, 14(2):257–299, 1998.

[44] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[45] T. Jian. An $O(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Transactions on Computers*, 35(9):847–851, 1986.

[46] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[47] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *IEEE Symposium on Foundations of Computer Science*, pages 81–91, 1999.

[48] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *ACM Symposium on the Theory of Computing*, pages 492–498, 1999.

[49] T. Kloks, D. Kratsch, and H. Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, 74(3-4):115–121, 2000.

[50] A. Koller, K. Mehlhorn, and J. Niehren. A polynomial-time fragment of dominance constraints. In *Annual Meeting of the Association of Computational Linguistics*, Hong Kong, 2000.

[51] A. local-ratio theorem for approximating the weighted vertex cover problem. Bar-yehuda, r. and even, s. *Annals of Discrete Mathematics*, 25:27–46, 1985.

[52] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[53] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction*. MIT Press, Cambridge, MA, 1998.

[54] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[55] K. Mehlhorn. Constraint programming and graph algorithms. In *International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, 2000. Invited Lecture.

[56] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In R. Dechter, editor, *Principles and Practice of Constraint Programming*, pages 306–319, Sep. 2000.

[57] B. Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25:239–254, 1985.

[58] B. Monien and E. Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22:115–123, 1985.

[59] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[60] G. L. Nemhauser and L. E. J. Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[61] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

[62] R. Niedermeier and P. Rossmanith. Upper bounds for vertex cover further improved. In *Symposium on Theoretical Aspects of Computer Science*, pages 561–570, 1999.

[63] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73(3–4):125–129, 2000.

[64] R. Niedermeier and P. Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36(1):63–88, 2000.

[65] R. Niedermeier and P. Rossmanith. Private communication, 2003.

[66] R. Niedermeier and P. Rossmanith. On efficient fixed-parameter algorithms for weighted vertex cover. *Journal of Algorithms*, 47(2):63–77, 2003.

[67] R. Paturi, P. Pudlak, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *IEEE Symposium on Foundations of Computer Science*, pages 628–637, 1998.

[68] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 359–366, 1998.

[69] R. Raz and S. Safra. A sub-constant error-probability low-degree test, and sub-constant error-probability PCP characterization of NP. In *ACM Symposium on the Theory of Computing*, pages 475–484, 1997.

[70] K. W. Regan. Finitary substructure languages. In *Structure in Complexity Theory Conference*, pages 87–96, 1989.

[71] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *National Conference on Artificial Intelligence*, volume 1, pages 362–367, Seattle, WA, 1994.

[72] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.

[73] J. M. Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical Report 1251-01, LaBRI, Université Bordeaux I, 2001.

[74] J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, 1990.

[75] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *IEEE Symposium on Foundations of Computer Science*, pages 605–615, 1999.

[76] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[77] R. Tarjan and A. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[78] A. C. Tucker. Coloring perfect $(k_4 - e)$-free graphs. *Journal of Combinatorial Theory*, pages 313–318, 1987.

[79] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[80] R. Yuster and U. Zwick. Finding even cycles even faster. *SIAM Journal on Discrete Mathematics*, 10(2):209–222, 1997.

[81] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 247–253, 2004.

[82] U. Zwick. All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms. In *IEEE Symposium on Foundations of Computer Science*, pages 310–319, 1998.